TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Igor Podgainõi 192469IAPM

A Unified Framework for Peer-to-Peer Applications

Master's thesis

Supervisor: Toomas Klementi
PhD student
Co-supervisor: Gunnar Piho
PhD

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Igor Podgainõi 192469IAPM

Ühtlustatud raamistik peer-to-peer rakendustele

Magistritöö

Juhendaja:	Toomas Klementi
	Doktorant
Kaasjuhendaja:	Gunnar Piho
	Doktorikraad

Tallinn 2022

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Igor Podgainõi

13.05.2022

Abstract

Bringing peer-to-peer applications to life can involve many difficulties. Usually, a developer needs to figure out which network architecture will work best for the software being designed, which technologies to use to traverse the NAT and how to make sure that the node is always accessible, no matter which network it is currently connected to. This is because the current Internet architecture has long moved away from being simple enough to allow uncontrolled and unrestricted communication between all nodes of the network. However, by abstracting an essential set of technologies into a unified structure and creating virtual layers on top of the Internet, it is possible to provide a compatible solution that would work automatically as if no obstacles had existed in the first place. The system in the Unified Framework lets the developer not care about how exactly to adapt to certain network situations, but simply write peer-to-peer applications as normal, and let the Unified Framework handle the background work, while at the same time maintaining compatibility with standard OS APIs and multiple programming languages.

This thesis is written in English and is 64 pages long, including 5 chapters, 15 figures and 2 tables.

Annotatsioon

Ühtlustatud raamistik peer-to-peer rakendustele

Peer-to-peer rakenduste loomine kätkeb endas mitmeid keerukusi. Tavaliselt peab arendaja selgeks tegema missugune võrguarhitektuur sobib kõige paremini loodava tarkvara jaoks, missuguseid tehnoloogiaid kasutada NATi ületamiseks ja kuidas tagada, et võrgusõlm oleks alati kättesaadav sõltumata sellest, missuguse võrguga ta parajasti ühendatud on. Selle põhjuseks on tõsiasi, et kaasaegse interneti arhitektuur on ammu lakanud olemast piisavalt lihtne tagamaks kontrollimatut ja piiramatut ühendust kõikide võrgusõlmede vahel. Abstraheerides põhilised tehnoloogiad ühetaolisse struktuuri ja luues interneti peale virtuaalsed kihid, on siiski võimalik välja pakkuda ühilduv lahendus, mis töötab automaatselt nagu mingisuguseid takistusi ei eksisteerikski. Ühtlustatud Raamistiku poolt pakutud süsteem vabastab arendaja kohustusest kohaneda konkreetse võrgusituatsiooniga ja lihtsalt kirjutada peer-to-peer rakendust tavaliselt, lastes Ühtlustatud Raamistikul teha taustatöö ning säilitades samal ajal ühilduvuse standardse operatsioonisüsteemi API-ga ja paljude programmeerimiskeeltega.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 64 leheküljel, 5 peatükki, 15 joonist, 2 tabelit.

List of abbreviations and terms

AES	Advanced Encryption Standard
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CA	Certificate Authority
CPU	Central Processing Unit
DAT	Distributed Address Table
DER	Distinguished Encoding Rules
DNS	Domain Name System
DDNS	Dynamic DNS
DHT	Distributed Hash Table
DLL	Dynamic-link Library
DTO	Data Transfer Object
E2EE	End-to-End Encryption
FFI	Foreign Function Interface
GCM	Galois/Counter Mode
GPIO	General-Purpose Input/Output
GRASP	General Responsibility Assignment Software Patterns
GUI	Graphical User Interface
GoF	Gang of Four
HMAC	Hash-based Message Authentication Code
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
ID	IDentifier
IDE	Integrated Development Environment
IP	Internet Protocol
IPv4	IP version 4
IPv6	IP version 6
ISP	Internet Service Provider

IV	Initialization Vector
IoT	Internet of Things
JSON	JavaScript Object Notation
LAN	Local Area Network
mDNS	Multicast DNS
MIT	Massachusetts Institute of Technology
NAT	Network Address Translation
NAT-PMP	NAT Port Mapping Protocol
NAT66	NAT IPv6-to-IPv6
OOP	Object-Orinted Programming
OS	Operating System
OSI	Open Systems Interconnection
P2P	Peer-to-Peer
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RNG	Random Number Generator
RSA	Rivest-Shamir-Adleman
SBC	Single-Board Computer
SHA1	Secure Hash Algorithm 1
SPoF	Single Point of Failure
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TODO	To Do
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
URL	Uniform Resource Locator
WAN	Wide Area Network
WebRTC	Web Real-Time Communication
WinAPI	Windows API
XML	eXtensible Markup Language
XMPP	eXtensible Messaging and Presence Protocol
XOR	eXclusive OR

Table of contents

List of figures

List of tables

1 Introduction

There are currently a lot of obstacles when trying to establish direct communication between two remote devices over the Internet. In the early days, it was possible to run a P2P (peer-to-peer) network effortlessly [1]. But today, developers of P2P applications may have problems. A P2P application can be defined as networked software, that in a technical sense does not care whether it acts as a server or a client [2], meaning, it would need to accept incoming connections from arbitrary locations when running on home and other restricted networks. When developing such software, a developer needs to either integrate all the necessary technologies that adapt to a specific network situation or let the user configure his network by himself in a way that would be accepted by the application [3]. These technologies need to be tightly intertwined with the networking code of the app, and all possible scenarios need to be taken into account. This is necessary because of the advent of NAT (Network Address Translation) which can prevent incoming connections from being accepted in some cases, as well as the need to rely on DNS (Domain Name System) to group IP addresses under a domain name [4].

A direct ”telephone line”-style channel should instantly be opened upon connection using the TCP protocol and its underlying IP protocol by routing it to the desired location designated by an IP address and port. With the Internet constantly evolving and having the structure that it does today, it is not possible to establish such a connection with all kinds of device combinations, just a limited set [5]. NAT-enabled networks combine several devices under one IP address, thus making it impossible for the networking hardware on the other end to know where exactly to connect, without explicit *port forwarding* on the source. Firewalls outside of the target device’s control can also exist along the way that would block the connection outright. Furthermore, assuming the ports are correctly forwarded, the IP address might change at some point due to it being *dynamic*, therefore, the same connection settings cannot simply be reused for an indefinite amount of time [4].

A typical scenario of the obstacles present during communication when hosting a service on today’s Internet can be described. In it, it is first necessary to find out the IP address that devices on the other end will see. It may not necessarily be known beforehand, because a NAT device may be present on the network that splits it into a private segment and the rest of the Internet, LAN (Local Area Network) and WAN (Wide Area Network), where the former indicates all devices within the NAT. To discover the WAN-facing IP address, one can either refer to the status page of the NAT device via a web interface [6] or use the help of special services on the Internet that simply report the WAN IP address back to the sender [7], which they can do because they would have the knowledge of it as recipients.

Next, the port number that the service is hosted on must be routed to a correct LAN IP address in the NAT device's configuration page, which due to ambiguity is not done automatically by the NAT device. This kind of routing is known as *port forwarding* [8]. Such a port must also not be blocked by additional firewalls that may be present on the network.

Once these prerequisites are met, it is now possible for other devices on the Internet to connect successfully to the host, however, its WAN IP address may still change as time passes, due to the phenomenon of *dynamic IPs* [4], so it may no longer be possible to establish new connections in the future. Typically what can be done to bypass this problem is to register a DNS hostname and set the A record to the currently used IP address. However, the usage of DNS can be problematic as well, as one would need to have the permission of the owner of the previous domain in the hierarchy to create domain names [9]. Even with this permission available, it may not always be possible as domain names can expire, which can cause hijacking [10], and changes do not always propagate instantly to users [11].

The final obstacle concerns ease of use. Many P2P frameworks that exist today have been implemented in a variety of programming languages. But they do not have a common structure or API, not to mention that they contribute towards a "lock-in" effect to a specific language [12]. It is necessary to have a framework that is truly "unified" in terms of its architecture and compatibility.

All these difficulties present a challenge for software developers that want to create a general-purpose P2P application.

1.1 Issues

Based on this, the specific set of issues that can arise during the development of P2P applications can be defined:

- The presence of NATs and firewalls on the network can prevent a direct connection from being established. No network should be excluded from being able to communicate directly, as long as it is connected to the global Internet.
- The impermanence or the lack of immediate knowledge of IP addresses used to identify a device on the Internet. This can lead to confusion and unavailability of services.
- The need to rely on centralized third-parties that report back the public (WAN-facing) IP or set up DNS hostnames. This can introduce unnecessary SPOFs (single points of failure).
- A lack of standards or compatibility between different existing P2P frameworks. This can make developers hesitate about which programming language to use for their application because of that factor.

1.2 Aims of the thesis

I propose the creation of a Unified Framework that would provide virtualization of OSI layers starting from the Network layer and up to the Presentation layer to abstract away all the technologies and details when developing P2P applications. With this system, it should be possible to solve the aforementioned issues while not requiring any major intervention by the target application developer, and being compatible with the standard POSIX-like networking API that is present in many popular operating systems today.

The Unified Framework should be portable to various platforms, including Unix-based. In this thesis, however, the focus will be on the Desktop platform running the Windows OS, due to its ubiquity [13] and ease of development. Also, the support for various programming languages should be included, including those targeting *machine code* [14] and not. This way, the Framework can be run on a wide range of devices. To achieve support for non-native programs (using higher-level languages), a ”bridge library” will be additionally created. The implementation given will allow the Python 3 interpreter to use the Unified Framework code successfully.

After the project has been completed, it will be tested using demonstration software and scripts that closely resemble real-world use cases.

There should be a number of different kinds of use cases for P2P applications and services that would be able to utilize the Framework. For example, as a developer of a document collaboration application, I want its user to be able to share his or her document publicly, so that other people can join in, read it and propose changes, without allocating any server storage space for that document. In another case, one might have developed a new machine-learning algorithm that takes up a lot of storage space, requires high-end hardware to run it and is currently proprietary. That developer can then host it in a distributed fashion (from multiple computers), where users can connect to the service, input the data that they want processed and get the output back. Finally, a developer wants to provide the users of Internet of Things (IoT) thermostat devices a way to connect to them on the go and adjust the temperature of the house while not being home.

Software of such kind [15] should be able to be developed without needing to take into account how the backend communication should function exactly, as if it is using the traditional client-server architecture [16], in any programming language of the developer's choice.

As the main result of the work done in this thesis, one should be able to use a fully-featured Unified Framework that is compiled in the form of a DLL file in a POSIX-compatible manner, such as in the example given in figure 1, or via the ”bridge library”.

```
UFNetwork::POSIX socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; // IP not used
addr.sin_port = 0; // port not used
socket.connect((struct sockaddr*)&addr, sizeof(addr));

socket.send(buffer, BUFFER_SIZE, 0);
```

Figure 1. Example connection code via the Unified Framework.

1.3 Structure of the thesis

This thesis is structured in 5 sections:

Section 1 introduces the problems as well as the goals of the thesis by proposing a solution. Section 2 discusses various technical aspects before the work can begin, specifies the review of relevant literature, explains the basics of the Unified Framework's design, as well as which technologies should be used during development and the work procedure. Section 3 describes in what form exactly has the result been achieved, as well as tells how to test and use the Framework. Section 4 evaluates the Unified Framework and outlines the developments for the future. The last section concludes the thesis.

The source code for the Unified Framework, demo applications, as well as various code examples, are available in the main project repository on GitLab[1].

[1] https://gitlab.cs.ttu.ee/igpodg/thesis-iapm

2 Methodology

This section explains the theory that is needed to be understood in the context of the project, reviews the relevant literature, as well as talks about with which paradigms and tools the final result will be created.

2.1 Background

Some concepts and technologies may need to be clarified before starting work on the project.

2.1.1 OSI model

The OSI model [17] [18] describes the separation of a communication system into seven distinct layers, where each is responsible for processing the data flow end-to-end from one device to another in terms of a specific context. The following paragraph provides an interpretation variant of this theoretical model applied to the Internet architecture so that its communication process can be inspected in detail, as it is complex. The diagram in figure 2 shows the model at a glance.

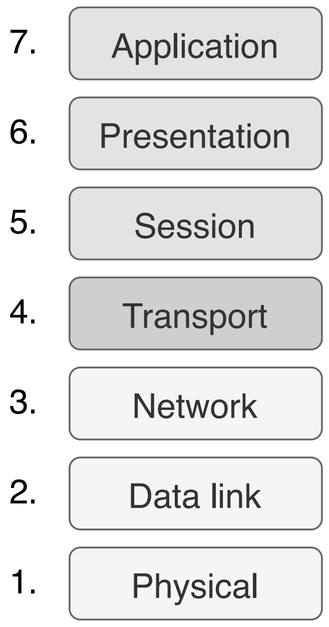


Figure 2. The OSI communication model.

The bottommost layer labeled as *Physical* deals with the processing of the data, including its re-encoding when needed, between hardware components, for them to properly ”understand” each other. The *Data link* layer deals with algorithms that allow the data to be transmitted and received by the hardware error-free and without collisions. The next layer, *Network*, is concerned with data routing in a logical space of locations represented by IP addresses. It also encapsulates data into *packets* for use by the *packet switching* system. The next layer, *Transport*, determines the ”nature” and various properties of communication to be used and agreed upon, for example, if it should be *connectionless* (UDP) or *connection-oriented* (TCP). This is also the lowermost layer that is accessible to application developers in common operating systems. The *Session* layer deals with multiplexing data addressed to the same device but meant for different applications. The differentiation is done with the help of *port numbers*. The *Presentation* layer encodes and decodes the data before the control over it is passed to the application, which can include encryption and decryption, for example. The final layer, *Application*, deals with the layout of the data in the context of the application itself. The application should know how to interpret the data at this level and what to do with it.

It is important to note that the OSI model was not used as a reference when the Internet architecture was first developed [19], as it did not yet exist, however, this fact does not affect the ability to apply this model to it. Also, the OSI model dissects the communication process between two devices into separate abstraction levels, where these levels are all active at the same time per unit of data, and different hardware or software components of the network are processing the same data differently.

2.1.2 IP addresses, ports and IPv6

There exist two common systems for addressing network devices: IPv4 and IPv6. IPv4 is the more widely used one of the two [20]. It allows to address $2^{32} = 4,294,967,296$ separate devices in theory, however, the amount of devices connected to the Internet today is a lot larger. Network Address Translation (NAT) is the technology that was used to mitigate this shortage. IPv6 on the other hand has a larger address space of approximately 18 quintillion devices. Due to this reason, IPv6 does not require NAT, so theoretically its usage can solve one of the issues posed by this thesis. However, Internet Service Providers (ISPs) and enterprises still prefer to use NAT with IPv6 to simplify routing and for other reasons [21]. For example, NAT66 (meaning, when both LAN and WAN IP addresses are

IPv6-based) can be used for *multi-homing* [22] so that one device can be connected to two or more networks simultaneously.

IPv6 will not be used within the Unified Framework due to its current lack of popularity with only approximately 40% of all devices globally using it[1].

Starting from this paragraph, the term *IP address* will refer exclusively to IPv4 addresses. IP addresses in the IPv4 system are 32-bit integer numbers that are typically represented in the *dot-decimal* notation, where every 8 bits are split using the ”dot” character and written out in the decimal form.

Various ranges or blocks of IP addresses are allocated to ISPs to give out to their customers. Some of them, however, are either reserved or have a special meaning rendering them unusable for addressing [23].

Port numbers are designed to identify a specific application on a device. They are 16-bit integers that are usually represented as a decimal number from 0 to 65535. Likewise, not all port numbers can be used for addressing [24].

The combination of IP address and port together describes the *location* of the remote device that one wishes to communicate with. This type of addressing will be defined as *location-based addressing*.

2.1.3 NAT and NAT traversal

As the amount of devices connected to the Internet is larger than the range of all available IP addresses, this technology effectively splits the network into two segments to bypass the problem. The Local Area Network (LAN) segment combines all devices that communicate with each other via the local NAT device, and the Wide Area Network (WAN) represents the rest of the network, in the context of which the entire LAN is visible only as one virtual device. The term *NAT* can also be used to mean a network that is currently using the NAT technology.

The LAN uses the same method of addressing as the rest of the Internet with the use of IPs, however, there is no one-to-one correlation between LAN and WAN IPs. The translation algorithm depends on the inherent type of NAT used: Full Cone, Restricted Cone, Port

[1] As of May 2022. More information: https://www.google.com/intl/en/ipv6/statistics.html

Restricted Cone or Symmetric [25] [26].

Typically, LAN IPs only span across ranges marked by IANA for *Private-Use* [27].

The use of Network Address Translation allows the ISP to only give out one IP address per customer, thus slowing down the resource exhaustion of IPs in the blocks available to them. To resolve ambiguity when it comes to knowing which LAN IP addresses of a specific device a certain WAN packet is intended for in case of client connections, the local NAT device keeps track of these mappings internally by maintaining a special table. A new entry gets added to the table whenever the first packet related to a certain connection gets sent from the LAN to the WAN. When the time comes to receive the response, the table entry of the needed mapping can be looked up [28]. However, this presents a problem with incoming packets that are received for the first time from the WAN in the connection context, since the required table entry would not yet exist.

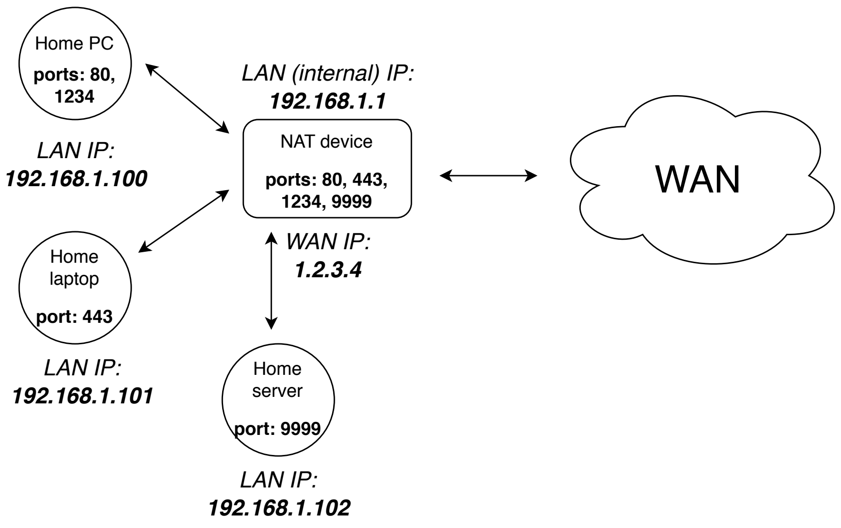


Figure 3. NAT device on a network with port forwarding.

The solution that was chosen for this is to provide a manual procedure known as *port forwarding*. Contrary to the naming used, this process concerns not just ports, but also LAN IPs. To use it, a user needs to visit the configuration page of the NAT device and input the desired LAN IP and port number, which would define an override NAT mapping based on that port. Some devices also provide a feature known as UPnP, which is a set of APIs designed for applications to communicate with the local network device [29]. UPnP includes an API allowing *port forwarding* as well via the *AddPortMapping* command [30]. From here on, the automatic port forwarding method using the UPnP API will simply be defined as UPnP.

It may not always be possible to access the configuration page of a NAT device due to the lack of privileges, and UPnP might not be available in the first place. In this case, the only other way to establish a direct connection where the initiator is connecting to a device within NAT is to employ *NAT traversal*. It is a set of techniques available for TCP and UDP Transport layer protocols that let two nodes ”trick” one or more NAT devices into allowing them to communicate directly with each other. The downside of this is that an extra device acting as a mediator between them would be needed. Such a device is known as a *Rendezvous* server or node [31].

The diagram in figure 4 shows an example of packets sent and received during a TCP connection that is traversed via NAT.

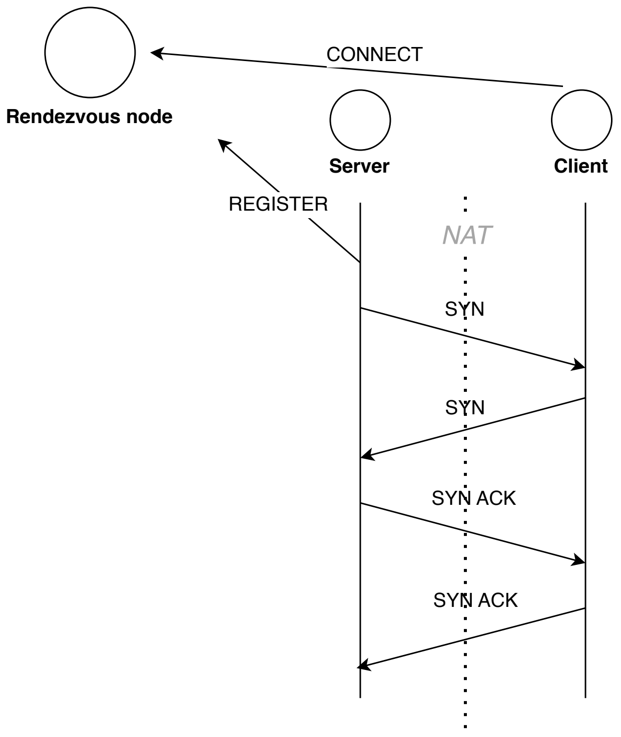


Figure 4. Example of NAT traversal via TCP hole punching.

2.1.4 POSIX networking API

Many operating systems nowadays, such as Windows [32], MacOS [33] or Linux [34], use a specific POSIX-compatible API as their primary one for networking operations. It is commonly known as *POSIX sockets* or *Berkeley sockets* [35]. The term *socket* refers to the context that describes a specific network connection.

A developer needs to follow a specific pattern of function calls to establish a connection and afterward to send and receive data. For TCP connections, it may look like the following: first, the *socket(...)* call is performed. Here is where it is possible to set parameters and flags that describe the type of the connection. The return value is a special handle that identifies a specific socket within the application. Next, a structure describing the parameters for outgoing connections may need to be initialized with the appropriate information. This is usually only used for server connections.

In this structure, the IP address field is set to a generic value to hint the operating system as to which *networking interfaces* such a connection should be ”listened” on. Typically, this is set to the constant *INADDR_ANY*, which is equal to the IP value of 0.0.0.0. For debugging purposes, the IP value of 127.0.0.1 is also often used, indicating the need to ”listen” only on the virtual *loopback* network interface, that is accessible only for connections across the same device. This value should be specified as a 32-bit big-endian integer. To apply the structure to the socket, the *bind(...)* call is used.

Afterwards, the *listen(...)* call can be used to switch the socket into server mode that would accept new incoming connections from yet unknown sources.

Finally, the *accept(...)* call is used for servers and *connect(...)* for clients to synchronously initialize a TCP connection. These two calls have differences in parameters required and the return values. The former returns a new socket handle and the structure describing the network location of the socket and doesn’t require any parameters, while the latter simply returns 0 in case of success, but requires that structure as an input.

As long as these requirements are met, it should be possible to implement a pseudo-socket system and API that mimics the standard one to maintain compatibility.

2.1.5 Machine code and the bridge library

Machine code, also known as *native code*, represents a program that runs directly using the instruction set of a computer CPU. C and C++ are examples of programming languages that compile directly down to native code. Languages such as JavaScript and Python are *interpreted* (at least to an extent), meaning they do not use native code for the execution of the application-specific code written in them [36]. Since the POSIX networking API in common operating systems does use it, these languages must invoke a compatibility layer known as a Foreign Function Interface (FFI) [37] between the OS and the interpreter to be able to call into it. This is where this phenomenon can be taken advantage of, where the Unified Framework can be inserted in-between this compatibility layer replacing the real API.

A special *bridge library*, also defined in this thesis using the term *shim*, can be created and *injected* into the process of the interpreter before any code begins execution, and all related functions *hooked* in order to redirect them to the Unified Framework [38]. Since the Framework is already compatible with the POSIX networking API, most functions inside the bridge library can be simply wrapped around. Hooking is complicated due to its technical methods, so projects exist that make it easier. For Windows, one such library is Microsoft Detours, which is available on GitHub and is licensed under the MIT license[1].

2.2 Literature review

It is also worth taking a look at some academic works that describe similar problems. I have discovered four such works that I would like to discuss.

2.2.1 The first work

The first one [39] talks about the ubiquity of IPv4 and the need of using NAT traversal under IPv4 due to the prevalence of NAT. Also, it admits that not always is NAT traversal possible due to ”a probability of failure”. In some cases, the usage of *relaying* might be necessary which is slower and the connection is no longer direct. Another way is to use UPnP or NAT-PMP for port forwarding. The Unified Framework will not use relaying due to it requiring additional indirection and having performance issues, but support for it can be technically added. UPnP will be provided as an option for port forwarding in the Framework.

The work references literature that describes three different ways that NAT traversal has been achieved in the past. The first one is considered by the authors as a *centralized* model, where the address of the Rendezvous server is known beforehand, so the nodes simply use it for traversal. The second method is dubbed as ”hybrid”, which seems to involve putting Rendezvous functionality onto standard nodes instead of having a separate node, as well as it requires the need to use public *proxy servers*. This approach seems to be infeasible for the Unified Framework again due to increased indirection. The third technique uses the Ethereum blockchain to distribute locations of nodes, instead of storing

[1] https://github.com/microsoft/Detours

them on Rendezvous servers. This would not be viable in the project present in this thesis due to the cost requirements.

The solution of the authors however is different from the three methods that they have referenced and is designed to be optimized for devices with low processing capabilities and power requirements, more precisely, for IoT (Internet of Things) devices. It involves putting Rendezvous functionality on ”gateway” nodes, which are network routers or similar equipment. They are calling their method DAT, which stands for Distributed Address Table, and is based on DHT (Distributed Hash Table). Additionally, encryption seems to take place in the DAT. The advantage of their method is that it optimizes for the *rate of churn*. If one ”gateway” node was to leave the DAT network, it would not affect the availability of stored node locations, but would only increase the latency.

This work seems interesting in that it proposes to move the use of Rendezvous functionality to ”gateways”, as well as that it uses a modified DHT. However, this method is probably infeasible practically due to network routers having firmware that restricts modification. But through cooperation with manufacturers, it may be a good idea to explore further.

2.2.2 The second work

The second work [40] tackles a data storage problem. Since data on the blockchain is not distributed but shared among all nodes, the authors propose using DHT instead. After they have finished developing the storage system, they propose applying it as an alternative to DNS for name resolution. The domain names would be stored in the DHT as data. Additionally, to prevent *domain squatting* they imply the necessity of fees. The idea is similar to the one proposed in this thesis for the Unified Framework. But instead of using domain names and storing them as data, arbitrary 160-bit identifiers will be used which represent entries in Mainline DHT.

The theory about various *blockchains* and DHT protocols is also included. Since the concept of a blockchain is not relevant to the Unified Framework, it will not be discussed. One of the mentioned projects is Handshake[1]. It seems to be an alternative to DNS, but instead of replacing it, it aims to coexist with it. The benefits are that domain names would not be removed by third parties and that it also provides some security features by ”avoiding poisoning risks”. This could be a project that is worth exploring.

[1] https://handshake.org/

The main part of the work talks about the network architecture, where blocks can be ”mined” on the DHT similar to a traditional blockchain. Again, the discussion of these parts is skipped because it is not relevant. At the same time, it is of interest to take a look at the Simulation part. The authors’ DHT-based name system estimates the *time complexity* compared to traditional DNS to be $O(2log_2N)$, where N is the number of currently available DHT nodes in the network, while DNS is defined to have the time complexity of $O(n)$, where n is the level of the hierarchy of a domain name. While this would indicate a generally slower performance of DHT-based lookup compared to DNS, it still confirms both the feasibility of such a system, as well as its relevance.

2.2.3 The third work

The third work [41] compares different symmetric and asymmetric encryption algorithms using benchmarking to determine the best combination. As a foreword, it acknowledges the need for E2EE (End-to-End Encryption) in modern communication contexts. Various messaging applications today utilize E2EE. In the Review section, this work lists some statistical analyses between users and concludes with the fact that many had been positively impacted after getting to know E2EE better.

Afterwards, the benchmarking results are outlined. It can be seen that AES performs best on different parameters such as encryption and decryption time, RAM (memory) and CPU utilization. The addition of asymmetric encryption in the form of RSA for the initial key exchange is negligible and does not affect the parameters to a significant extent. The Unified Framework uses a TLS-like protocol for key exchange during data encryption, which in turn uses AES in conjunction with RSA.

2.2.4 The fourth work

The final work [42] tackles the problem of home appliance system availability.

The work mentions related literature that proposes the usage of ”control systems”, meaning centralized devices. It also mentions that due to physical length constraints of cables it may be difficult to place such devices in a certain area.

It is important to note that the equipment owned by the authors, in this case, is located on a home network that uses *dynamic IPs*, meaning the WAN IP address given away by the ISP can vary per each renewal. The costs can seemingly increase in case one wishes to acquire a *static IP*. This is the reason why the authors acknowledge the issue and try to resolve it.

The work describes the physical layout of the equipment: a Raspberry Pi SBC (Single-Board Computer) and an actual appliance control circuit connected to it via GPIO pins. This combination is replicated several times across the suites of the building. The different Raspberry Pi devices seem to also connect in a mesh-like network, but only one of them is used as an *exit node* to the WAN.

This describes a scenario of the need to use DDNS on a low-powered device. The method works, and the authors note that it is inexpensive, however it relies on a specific DDNS service, which can be a single point of failure. In case something goes wrong with it, the entire management of the house can be inaccessible remotely.

2.2.5 Conclusion

To conclude, these works demonstrate the relevance of the problems, as well as the solutions that they proposed. NAT traversal may still be needed even though IPv6 exists since it is not common enough yet. Also, it is technically possible to use DHT instead of DNS to address devices based on an identifier. End-to-end encryption is prevalent and does not consume an exorbitant amount of computing resources. There exist scenarios in which servers are located in home networks and may need remote access functionality. Finally, IoT (Internet of Things) devices are prevalent nowadays, so having the Unified Framework be compatible with native code comes hand in hand with this trend.

2.3 Overview

With the relevant theory explained, it is now possible to define how the Unified Framework should function.

The core part of the Framework is the class that is centered around the POSIX networking API. The methods should have a similar or same interface as well as functionality to it. This is needed to provide compatibility for various programming languages and familiarity for developers, and also because the rest of the Framework's code would then be able to reuse it for its own needs. In terms of the Framework's location according to the OSI model, it should span across the Session and Presentation layers and replace the concept of network location determined by an IP address and port with a single identifier to achieve *identity-based addressing* without the use of DNS. This means that some identifier is used to pinpoint a specific device instead of its location.

Another vital component would be the various NAT traversal methods that allow direct communication to take place no matter which network each of the devices is currently located on. Finally, data encryption should be performed at all times during communication no matter which Application layer protocols devices use to ensure adequate privacy.

To summarize, the Framework's functionality can be divided into four parts: the facilitation of direct connections between Internet devices by means of traversing the NAT, the substitution of location-based addressing with identity-based addressing, using cryptography and encryption to achieve confidentiality of data and the maintenance of compatibility with the POSIX networking API and multiple programming languages.

2.4 Tools and technologies

The technologies that were used during development are listed as follows:

- *Mainline DHT*, also known as BitTorrent DHT [43], is a public service that provides a simple distributed database for key-value pairs, where the key is some 160-bit integer identifier and the value is a network location represented by an IP address and port. The distribution of the key range using XOR distance allows for a delegation of responsibilities among participating nodes in a simple way to achieve its distribution.
- *UPnP*, more specifically *UPnP port forwarding* [30], is a standard API included in many NAT devices that allows developers to automate the process of forwarding LAN IP addresses and ports for a device to be able to accept arbitrary incoming remote connections for the first time.
- *TCP hole punching* [31] is a method to achieve NAT traversal over TCP.

Likewise, a list of tools is specified as follows:

- *OpenSSL*[1] is a popular cryptographic library. In the Unified Framework, it will be used to provide symmetric encryption using the AES-192-GCM algorithm, asymmetric encryption using the RSA-2048 algorithm, as well as to generate 160-bit codes using the HMAC-SHA1 function.
- *pugixml*[2] is an XML processing library for C++. It will be used for interaction with the UPnP API on NAT devices. It is licensed under the MIT license.
- *WinAPI*[3], the main API of the Windows operating system, will be used for its networking features, random number generation, GUI capabilities and injection of code into processes.
- *wxWidgets*[4] is an advanced GUI framework. Its extended capabilities will be used during the creation of demonstration applications.
- *Microsoft Detours* is a library designed for redirecting functions inside a running process. It will be used in the project for the Unified Framework shim.

The Unified Framework itself is developed in the C++ programming language because it is compiled down to machine code, which means it can be made to work with both native and non-native applications.

2.5 Work procedure

The software design and development techniques chosen for the Unified Framework were *modular programming* [44] and the *bottom-up approach* [45].

The entire project is divided into modules. As soon as one module has been developed and tested, the development then took focus on another part, until the whole project could be brought to life by connecting the pieces.

The current implementation has been developed in Visual Studio for the Windows platform. The source code is structured in an Object-Oriented Programming (OOP) style with inner classes. The end result is compiled down to a DLL file named *uf.dll*.

[1] https://www.openssl.org/
[2] https://pugixml.org/
[3] https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list
[4] https://www.wxwidgets.org/

3 Main outcomes

This section describes the technical aspects of the project, namely its design, the various implementation details, network architecture as well as examples of use. Also, demonstration applications will be presented for testing purposes.

3.1 Designing the Framework

This section outlines the different modules and components that are integral to the core functionality of the Unified Framework.

3.1.1 Classes

The functionality of the codebase is split across multiple classes, several of which are nested. Table 1 lists the few key classes that influence the Framework's behavior as well as describes their purpose. The dot character indicates a nesting hierarchy, meaning that the class on the right side of the dot is a part of the class to the left of it.

3.1.2 POSIX-like interface

The UFNetwork.POSIX class provides the familiar POSIX-like networking interface with the following methods: *htons(...)*, *ntohs(...)*, *getsockopt(...)*, *setsockopt(...)*, *ioctl(...)*, *getsockname(...)*, *inet_pton(...)*, *inet_ntop(...)*, *bind(...)*, *listen(...)*, *accept(...)*, *connect(...)*, *send(...)*, *recv(...)*, *shutdown(...)*, *close()*. They are available in OOP and non-OOP form, so it is possible to initialize the class not only using a constructor, but also to get the socket handle directly using the *socket(...)* method. It is also possible to get the socket handle later from the class instance using the *getSocket()* method.

Most of these methods simply wrap around the existing networking APIs that are provided by the OS, but there is one major difference. The socket handle values no longer correspond to the system ones. Instead, the Unified Framework manages its own socket handles along with an additional context, which is defined by the *Context* structure inside the class.

Table 1. Class listing of the Unified Framework's codebase.

UFCrypto	This class is responsible for cryptographic operations.
UFNetwork	This class contains functionality to process network-related tasks. Excluding its inner classes, it only contains a few base variables and methods, such as the ones intended to initialize buffers and the OS's networking API.
UFNetwork.Node	This base class defines a network node to be used within the Unified Framework. It is in charge of maintaining the storage and loading of cryptographic keys, setting the base node parameters and initializing the node.
UFNetwork.UPnP	This class manages the API built into NAT devices that is designed to forward LAN IPs and ports to the WAN.
UFNetwork.DHT	This class implements the client for the Mainline DHT network that is normally used by BitTorrent. It can initialize the node cache, as well as find a location value from a specified key (node ID).
UFNetwork.DHTAnnounce	An extension of the previous class that can write (announce) data to the DHT.
UFNetwork.Rendezvous	This class provides a Rendezvous node for others in case they require one as part of a NAT traversal protocol.
UFNetwork.POSIX	The primary class of the Framework that provides a POSIX-like API for direct use by shims or developers.
UFUtil.DebugProcessor	An abstract class that is intended to define the procedure of how the debugging output should be processed.
UFUtil.DebugLogger	A class that implements logging of the debugging output to the console.

The constructor as well as the *socket(...)* function contains some additional non-standard optional arguments. The first one defines the *connection behavior* of the node. The purpose of this argument is explained further below. The second parameter allows one to specify an existing instance of the UFNetwork.Node class, since its features are provided in UFNetwork.POSIX not via *inheritance*, but with the use of *composition*.

The connection behavior of a POSIX server or POSIX client node determines the method to be used when establishing a direct connection. Three different methods are available: *TRAVERSE_USING_RENDEZVOUS* specifies that the connection should always be performed using NAT traversal via any available Rendezvous nodes. *ASSUME_PORT_OPEN* is a method that always assumes that a direct connection is possible by default. Finally, *FORWARD_USING_UPNP* first forwards LAN IPs and ports before establishing any connections.

In the current version of the implementation, the connection behavior variable is ignored and the *TRAVERSE_USING_RENDEZVOUS* value is always assumed.

The methods *accept(...)* and *connect(...)* can be considered special in that they perform additional operations as part of their execution. First, the *initializeNode()* method of the UFNetwork.Node class is called, which loads and generates the required cryptographic keys, as well as *bootstraps* and *announces* the node to the DHT. Then, depending on the current connection behavior, additional initialization is performed before the control is passed back to the application.

The methods *send(...)* and *recv(...)* also perform encryption and decryption along with the standard processes.

3.1.3 Data encryption

Two sets of keys are defined as part of the data encryption engine in the Unified Framework: the AES-192 *application key* and the *secret key*, which internally is an RSA-2048 keypair. Most cryptographic features are provided by the OpenSSL version 3 library, except ones such as random number generation, which is achieved using the WinAPI function named *RtlGenRandom(...)* (a pseudo-RNG).

The application key can be of an arbitrary value, so the output of the pseudo-RNG is simply used for its initialization. Since RSA keys need special formatting, the secret key is created using OpenSSL and stored inside a binary buffer in the ASN.1 DER format.

Whenever a call to the *send(...)* function is made within the UFNetwork.POSIX class, the packet is encrypted using a specific key with the AES-192-GCM algorithm before it gets sent to the destination. The header of the encrypted packet includes the size of the encrypted data and the initialization vector. Likewise, at the invocation of *recv(...)*, the packet is first decrypted before it is passed back to the application. In case the packet is found to be corrupted, then an exception is generated, which is considered to be a ”socket error”.

These keys also have a secondary purpose: the application key is used to provide a shared application context, as in, anyone who has access to the application can read that context’s data and find its nodes, and the secret key is used to establish a private context, where only two parties are supposed to be aware or access its data.

Since the secret key does not have an AES-192 form, a temporary key needs to be agreed upon between two parties using the provided RSA-2048 pair. This is known as *key exchange*, and the protocol for it in the Unified Framework is similar to TLS [46] and looks like the following:

- The client sends its public component of the secret key to the server. It is no problem if it does not change and is shared with multiple parties, as it is intentionally public.
- The server generates a new temporary AES-192 key and encrypts it using that public key then sets it as the one to be used for the current TCP connection.
- The client node receives the data back and decrypts it using its own private key to retrieve the AES-192 key that is ”wrapped” inside.
- An encrypted communication channel has now been established. The socket handles are given back to the application code and execution can continue.

One difference between TLS and this protocol is that Certificate Authorities (CAs) are not used. This is due to the requirement of not relying on a centralized third party.

To facilitate the generation of keys by a developer, a static method is available in the UFCrypto class named *generateKeysToDisk(...)*. After the execution of this method, the necessary encryption keys are stored on disk with the filenames *app.key* and *secret.key*. The former contains the aforementioned application key, which should be the same between all instances of the application across the network, and is required for the Framework to function under all circumstances. The second file is optional and is only needed to establish potential private communication with another node.

The encryption features are provided by the OpenSSL 3 library.

3.1.4 Node identifiers

As it was mentioned previously, the encryption keys are also secondarily used in the Unified Framework to establish various contexts. This is implemented using the HMAC-SHA1 [47] [48] function, which takes two parameters, a key and a message, and produces a *message authentication code*. This code is only used in the Framework to substitute a node ID for the DHT, and it is used instead of a hash function because it has two inputs instead of one.

Only two types of node IDs exist within the Framework: *public* and *private*. The public node ID is derived from the application key using HMAC-SHA1(k, m), where k = the application key itself and m = the *prefix* that describes a specific node role.

The output value is 160 bits, the same length also used for entries in the Mainline DHT, so it suits it well. The private node ID is similarly a derivative of the application key using HMAC-SHA1, but here the prefix is not used anymore and the m variable contents are replaced with the public component of the secret key. As it is not possible to calculate the private node ID value without having access to a specific public key, which should be different between all instances of the application, it can simply be given away in some manner to another party that wishes to establish a two-way connection, for example, via a third-party messaging channel or in person.

Similarly to data encryption, HMAC-SHA1 operations are performed with the help of the OpenSSL 3 library.

3.1.5 DHT

The used variant of DHT is a distributed table of key-value pairs, where the *key* is some 160-bit identifier and the *value* is a location represented by some IP address and port of a service. It is an essential part of the Framework and is a starting point for beginning any kind of network communication via the Unified Framework. The DHT is being used, as the key goal of the Unified Framework is to replace location-based addressing with identity-based addressing, but since it is merely an abstraction, the location still needs to be known by the application eventually.

The classes UFNetwork.DHT and UFNetwork.DHTAnnounce represent a Mainline DHT client. When the node is initialized using the *initializeNode()* method of the UFNetwork.Node class, the node IDs are first generated from the files *app.key* and *secret.key*, then stored in memory. Since the private node ID is optional (not always there is a need for private communication contexts), it will not be set in case the file *secret.key* is not currently present on the file system. Afterwards, the DHT is bootstrapped, as in, prepared for functioning, then if the current node has server-like characteristics, meaning it needs to be found by others, the available node IDs along with the current WAN IP address and a random port are *announced* (written) to the DHT.

In order to interact with and bootstrap the DHT in the first place, it is necessary to use an initialization node with an already known location. Such a node is provided publicly by Rainberry Inc.[1]: *router.bittorrent.com* at UDP port 6881. This value is hardcoded into the Framework and can introduce a single point of failure. However, since a list of random DHT nodes is generated during the bootstrap process, it should be possible to supply it with the application distribution package or cache it on the device. This is not done in the current implementation.

Finding a node by ID is done similarly to announcing, but in this case, an extra parameter is needed that determines the ID of the node to be looked up.

All nodes of the same role are considered to be equivalent, so they are represented by the same public node ID in the DHT. Private node IDs do not make a distinction between node roles.

3.1.6 UPnP port forwarding

The UFNetwork.UPnP class provides automatic *port forwarding* via NAT devices. This private class does not represent a node, and it is only meant to be used internally by other components of the Framework, such as, when the *FORWARD_USING_UPNP* connection behavior is set.

The interface of the class is quite straightforward. It allows for enumeration of available network cards on a computer and performing UPnP port forwarding in case a NAT device is present under that network card. It is possible to forward any port for the TCP and UDP

[1] The company responsible for developing µTorrent. More information: https://rainberry.com/

protocols. The LAN IP is calculated by the NAT device automatically, it is the IP address that the current device uses to connect to the network.

3.1.7 Rendezvous

Rendezvous is a helper node that is meant to always be active on the network within a certain application context defined by the application key. If at least one such node is present, that means NAT traversal will be possible for nodes within such a context. This node is meant to be active for as much time as possible to increase availability. The prerequisite for hosting is that it should not be under NAT by itself.

The UFNetwork.Rendezvous class internally calls UFNetwork.POSIX methods, because it needs to support encryption during communication as well.

The node only has two commands to complete its operations correctly: *REGISTER* and *CONNECT*. The *REGISTER* command is meant for registering or announcing a server node that intends to accept new incoming connections, so the server sends a message to all Rendezvous nodes currently present in the context containing the node ID of the server and its listening port number. Later, when a client node wishes to connect to the server, it queries these Rendezvous nodes using the *CONNECT* command, where it specifies the server's node ID and the source port bound to the client using the *bind(...)* call, as per the TCP hole punching algorithm. In the case of matching values, the Rendezvous node produces a successful scenario that allows the server and the client to directly connect.

To launch a Rendezvous node in code, it is sufficient to simply initialize the class and call the blocking *runServer()* method.

3.2 Implementation details

Some details of the Framework's design should also be clarified. This section lists the ones that can be considered the most important.

3.2.1 Node roles and prefixes

The previously mentioned prefixes of a node are arbitrary values that are only used for node ID generation. They are 32-bit values that all nodes of a certain type have. The role determines the purpose of the node in relation to the network architecture, of which there are currently three: Rendezvous, POSIX server and POSIX client.

In the implementation, these values have been randomly generated and are located in the UFNetwork.Node class as static constants.

3.2.2 DHT bootstrap process

As mentioned before, in the Unified Framework the DHT client provided by the UFNetwork.DHT class has a *bootstrap* process. During it, a list of random DHT nodes is collected into a list using a randomly generated DHT ”node ID”, and only using these nodes are other operations done. This is implemented in this way to increase anonymity, but also because such a list of random nodes can technically be saved into a file or cached in memory for later use.

There is also a difference between the bootstrap initialization node and bootstrap nodes in general. The initialization node is a hardcoded value that always points to the same location. It is not possible to connect to the DHT without it unless the locations of other nodes are known beforehand.

The class UFNetwork.DHTAnnounce is a subclass of UFNetwork.DHT. In addition to the bootstrapping process and the ability to find locations by node ID, it provides the extra feature of *announcing* a new entry into the distributed table. For this, a secondary bootstrap operation is needed in the current implementation.

Also, there is an overlap between the terms used within the Unified Framework and Mainline DHT. The latter uses the terms ”node ID” and ”info hash”. The info hash corresponds to public and private node IDs in the context of the Framework.

3.2.3 Establishing private communication

All data transmission is end-to-end encrypted between the different nodes in the network architecture of the Unified Framework. By default, the shared AES application key is used for this purpose. To switch to private communication using the secret key, a combination of a few steps needs to be performed first.

Firstly, the values *s_addr* and *sin_port* of the *sockaddr_in* structure need to be set to hexadecimal values 0xFFFFFFFF and 0xFFFF respectively (representing a fake IP 255.255.255.255 and port 65535). These dummy values indicate to the Framework in a POSIX-compatible way that further communication with the current socket should be private. In order to connect to another private node, one would need to know its private node ID. Since there is no compatible way to specify it, a small GUI dialog box shows up for POSIX client nodes as soon as the *connect(...)* function is called. For server nodes, the private node ID is generated automatically from the *app.key* and *secret.key* files that should be present on the file system. It is possible to use the static method *calculatePrivateNodeIdFromKeyFiles(...)* in the UFCrypto class to get the private node ID for the current node from the files.

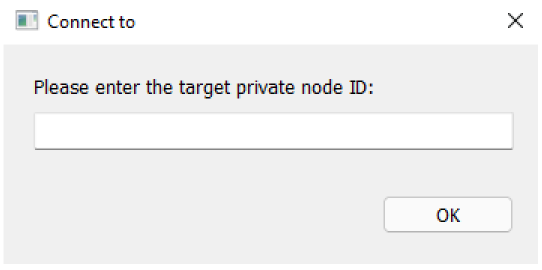


Figure 5. GUI dialog box provided by the Framework.

An improvement to the Framework would be to let the developer use a method that would attach the private node ID that he wishes to connect directly to the socket instance. However, this may undermine the maintenance of POSIX compatibility.

If the private communication mode is enabled, the key exchange protocol is performed after the TCP channel has been established, but before the function returns control to the application.

3.2.4 Inner workings of the bridge library

The process of the bridge library's functioning can be divided into two parts: injection and function hooking (redirection).

The injection part is responsible for making sure that custom code can be run in an application of choice. To achieve this, a special *injector* program is used. It uses a combination of WinAPI functions (known as the *CreateRemoteThread* method) to start a process with the desired command-line arguments with the bridge library performing whatever its operations are before the application-specific code begins executing.

The redirection part takes place in the DLL file (*ufshim.dll*) that has been injected. It uses the various API calls provided by the Microsoft Detours library to redirect all invocations of the POSIX networking API inside the application to the Unified Framework's virtual POSIX-like API instead. After that, the application should be able to continue its work not suspecting anything different. Due to the UFNetwork.POSIX class in the Unified Framework already being similar enough to the real POSIX networking API by design, most *hooked* functions are simply wrappers around that class.

The bridge library is intended to be universal and support different kinds of applications. However, since some of them may use system APIs in an unconventional way, the library may need additional patches over time.

The source code for the bridge library and the injector program are available in this repository on the university GitLab[1] and is licensed under the MIT license.

3.3 Network architecture

Figure 6 shows the placement of virtual layers provided by the Unified Framework in relation to the theoretical OSI model.

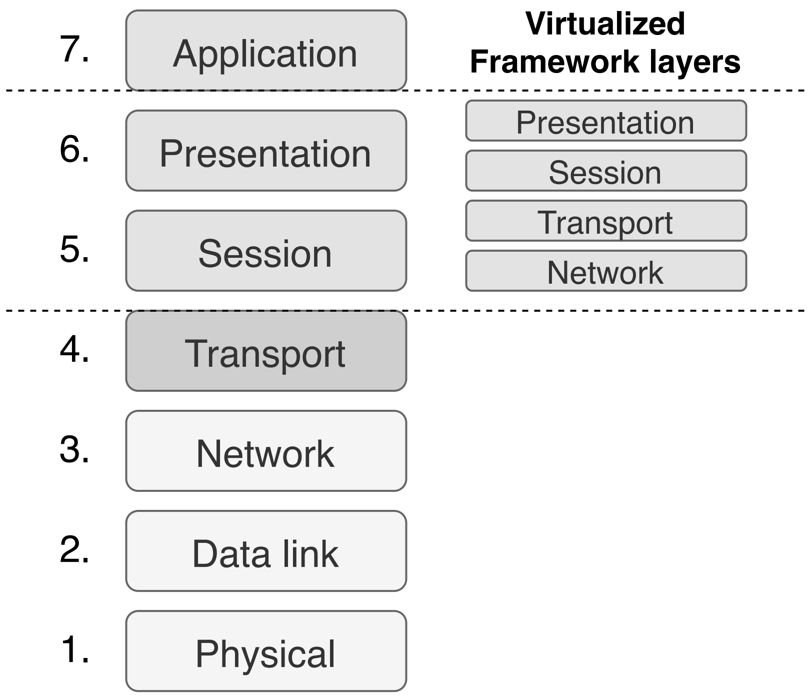


Figure 6. Unified Framework's layers in relation to OSI.

To explain in more detail the meaning of this placement, the description of each virtual layer in this context can be described. The virtual Network layer provides identity-based

[1] https://gitlab.cs.ttu.ee/igpodg/detours-example/-/tree/ufshim

addressing using public and private IDs between nodes. The virtual Transport layer performs NAT traversal or UPnP port forwarding (if necessary), then simply passes through to the real Transport layer, which in this context is TCP communication. The virtual Session layer maps dummy port numbers to real ones that are written to the DHT, as well as provides a POSIX-compatible networking API. Finally, the virtual Presentation layer provides data encryption and decryption capabilities. Afterwards, it is up to the application how to handle the remaining high-level layer.

It is also possible to outline the network architecture as a topology. Various scenarios of network layouts will be presented.

The most simple topology shown in figure 7 of a network of an application using the Unified Framework has a total amount of two unique nodes: one POSIX server and one POSIX client node. This configuration can be achieved in case neither of the nodes requires NAT traversal. The nodes can then communicate privately using either the application or secret key. However, their privacy may be undermined if a new node joins the network later if the application key is used. Since it is shared, all the information communicated between the first two nodes will be easily accessible by the newly joined node.

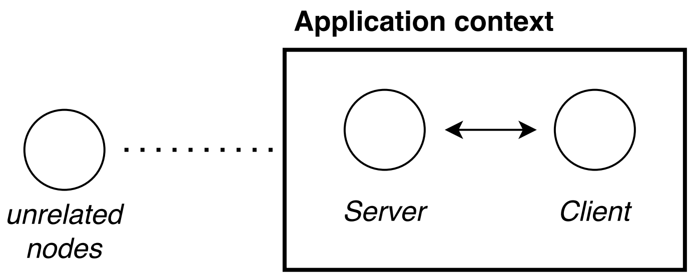


Figure 7. Example of simple network topology.

A more complex topology scenario takes the previous one as a base and introduces one additional Rendezvous node.

In the example in figure 8, the POSIX server requires NAT traversal due to network restrictions, while the POSIX client does not. In order to begin communication, the client connects first to the Rendezvous node, and only then to the server node. Both of these nodes will still have to rely on the help of the Rendezvous node anyway because the server is the one that needs to accept arbitrary new incoming connections. If the requirements were reversed, however, then using the Rendezvous node would not be needed.

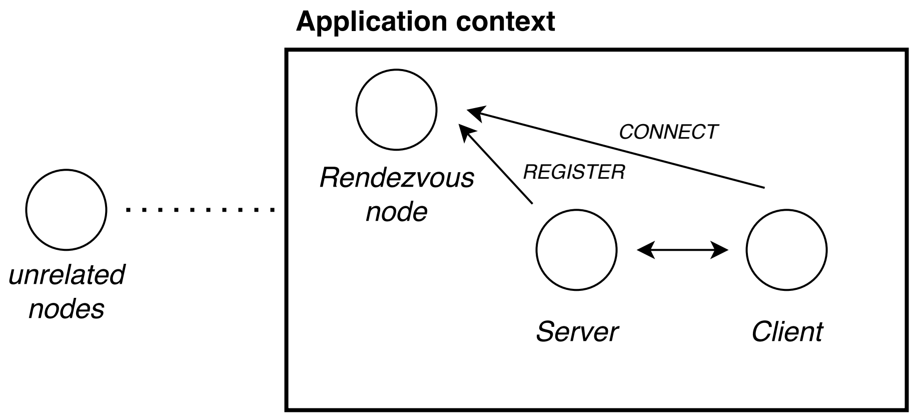


Figure 8. Example of complex network topology.

The final scenario in figure 9 shows the presence of three Rendezvous nodes, two POSIX server nodes and one POSIX client node. All POSIX nodes in this case require NAT traversal. Since there is more than one Rendezvous node, the servers have to register themselves with all such nodes they can discover. Afterwards, the client then has to query all Rendezvous nodes until it finds a match. By having three of such nodes, it is possible to avoid single points of failure, but this procedure can, unfortunately, introduce *bottlenecks* and the overload of traffic.

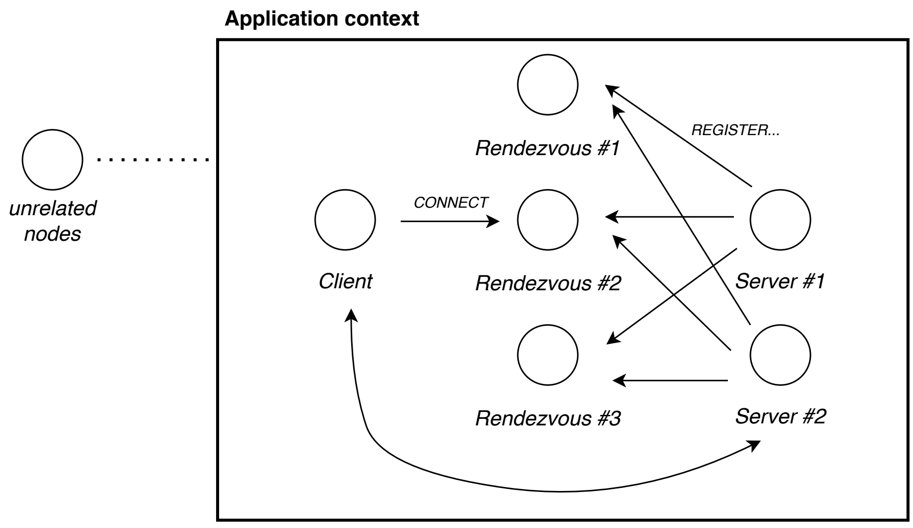


Figure 9. Example of another complex network topology.

The sudden unavailability of all *Rendezvous* nodes at once would allow existing NAT-traversed connections to continue operating, but no new such connections will be able to be made until another node becomes online.

In the current implementation, only registration with the first Rendezvous node that has been found is performed.

3.4 Testing the Framework

To test the correctness of all parts concentrated in the design of the Unified Framework, as well as proper operation of the implementation developed in C++, a few demo applications have been created. This method is similar to *integration testing* [49], and it was chosen because the structure of the Framework is complex and intertwined, so unit tests would not have worked as ideally for this purpose.

3.4.1 ChatApp

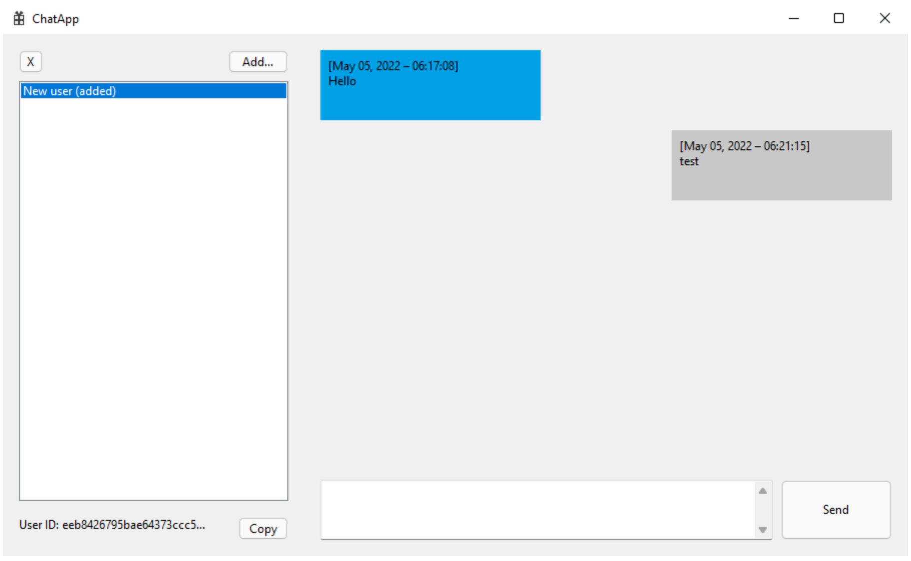
The first demo is a messenger application named ChatApp that closely simulates real-world chat software. While it has no special parts in it, other than the inclusion of the Unified Framework, it can demonstrate the complete behavior of the Framework from beginning to end. The part that is used is the UFNetwork.POSIX class from the Unified Framework. Both POSIX server and POSIX client functionality is present. Additionally, it includes a few other required methods, such as the one to retrieve the private node ID from local key files.

ChatApp is written in the C++ language, which targets native code, thus demonstrating the ability of the Unified Framework to get integrated into the software of such kind.

ChatApp does not have any knowledge of the virtualized communication layers provided by the Unified Framework, and it believes that it is either listening on a certain IP address and port to accept incoming connections or connects to another device directly via raw TCP and exchanges *plain text* packets, which is not actually the case.

The application is GUI-based and has two *views*: the initialization screen and the main screen. The former is needed because the Framework needs to initialize before allowing the software to establish any kind of network communication. This can be seen as a downside. A command-line window is also open for debugging purposes. If a Rendezvous node is not present on the network and is not using the same application key as ChatApp, then the initialization will fail and report a ”socket error”.

On the main screen, the User ID can be seen, which is internally a private node ID of the POSIX server component inside the application. This ID is calculated automatically from the *app.key* and *secret.key* files. The *Add...* button will invoke the *connect(...)* method immediately. However, because the private communication mode is used, a GUI window



New user (added)

User ID: eeb8426795bae64373ccc5...

[May 05, 2022 – 06:17:08]
Hello

[May 05, 2022 – 06:21:15]
test

Figure 10. The main screen of ChatApp.

supplied by the Framework will first ask the user for the target ID to connect to.

The following scenario can be set up to test the Unified Framework via this demo application: one Rendezvous node with debugging output enabled and located on a network that does not use NAT, as well as two ChatApp instances on separate networks that each use NAT. Two unique secret keys should be located in the respective directories of ChatApp. If one of the instances can successfully connect to the other via the user ID, and both of them can send messages to each other, then it means that the communication has been successful.

The debug command line window of ChatApp should also not show any errors and the debugging output of the Rendezvous node should indicate a successful traversal. In this case, the test can be considered ”passed”.

To further verify that the encryption methods are valid, it is possible to use software such as Wireshark[1] to *dump* the packets going back and forth and attempting to decrypt them using the appropriate keys.

3.4.2 Python demos

To perform further additional testing with a different programming language that does not use native code, Python 3 was chosen due to its quite standard internal use of the POSIX networking API.

[1] https://www.wireshark.org/

Unlike the previous demo, it is not sufficient anymore to simply import the Unified Framework DLL file and interface into the code. While there are ways to execute DLL functions within Python code, this would require a heavy modification of the code layout. Instead, the Framework can be embedded through the use of a bridge DLL file, while maintaining the same kind of API virtualization.

To inject the Unified Framework into a Python script, the following four files should be present in the path of execution: *inject.exe*, *ufshim.dll*, *uf.dll* (the Framework DLL itself) and *app.key* (the shared application key). Then, the keyword *inject* can simply be prepended in a command-line window to whatever is the program to be launched, for example, in this way: *inject python script.py*.

Three different Python scripts are presented for demonstration. The first one named *chatapp_send.py* is intended to confirm that it is possible to connect to a ChatApp instance within Python. While the previous demo is still running, the script should be executed with the following command line: *inject.exe python chatapp_send.py*. After the input of the target ID and the initialization, the GUI window of ChatApp should display the message ”*Hello from Python!*” from a new pseudo-user.

The other two scripts represent a tiny Flask microservice [50] and an HTTP client that connects to it. In this scenario, a Rendezvous node is also present in the network.

On two remote devices at different locations, the scripts *server.py* and *client.py* should be run through the injector one after another. The test can be considered ”passed” if the client shows a JSON object on the screen containing the server’s local time. No additional GUI windows should show up in the process, since compared to the previous demo, this one uses public communication via the application key.

Using these demonstrations, it is possible to confirm the successful integration of the Unified Framework into Python code for compatibility. The current implementation of the Framework has been tested with Python version 3.10.2.

3.5 Using the Framework

The Framework is designed to be used in two ways: being dynamically linked directly into a native code application or injected via the bridge library.

3.5.1 Dynamic linking

For the first approach, the target application needs to support linking to C++-style symbols (names). The usage in C++ applications will be explained.

First, the Unified Framework needs to be compiled from source code. Currently, it is a Visual Studio-based project, so one needs to be able to use this IDE for compilation. Upon opening the file *uf.sln*, the subproject *uf*, which represents the Framework, can be built for x86 or x64 architectures. Support for ARM64 has not been added to the current implementation.

This produces the necessary files *uf.dll* and *uf.lib* for subsequent dynamic linking. Next, the appropriate project for the desired C++ application should be configured to link to these files during compilation. In Visual Studio this can be done by navigating to ”Properties”, ”Configuration Properties”, then ”Linker”, ”Input” and ”Additional Dependencies”. Here is where it is possible to specify the location of the *uf.lib* file.

Now it is necessary to *include* the *uf.h* header file into the source code. This file is available in the repository in the ”*uf_test*” folder. After this, it should be possible to use the API of the Framework. The static method *UFCrypto::generateKeysToDisk(...)* should first be called to generate the new application key and optionally, the secret key. The arguments of the method determine which keys need to be created. Finally, the appropriate code depending on the node type needed can be inserted into the application.

```
#include "uf.h"

int main() {
    // generate the application key and output the file to disk
    UFCrypto::generateKeysToDisk(true, false);
    return 0;
}
```

Figure 11. Short application that generates a new application key.

The subproject *uf_test* in *uf.sln* provides an example C++ application that imports the Unified Framework.

There are not many differences in the actual usage of the Framework compared to the POSIX networking API, since it was intentionally designed to be compatible with it. Here are some examples of code that can be used by developers:

The code snippet in figure 12 shows the creation of a new socket in the object-oriented style, the addition of a debug logger, the population of the *sockaddr_in* structure, and finally the acceptance of new connections as a server.

```
#include "uf.h"

#include <iostream>
#include <cstdio>
#include <stdexcept>

char buffer[8192];

int main() {
    UFNetwork::POSIX ufSocket(AF_INET, SOCK_STREAM, 0, FORWARD_USING_UPNP);
    ufSocket.setDebugProcessor(new UFUtil::DebugLogger);

    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = 0;
    ufSocket.bind((struct sockaddr*)&addr, sizeof(addr));

    ufSocket.listen(SOMAXCONN);
    struct sockaddr_in incomingAddr;
    int incomingSize = sizeof(incomingAddr);

    std::cout << "Listening...\n\n";
    while (true) {
        socket_t incSock;
        if ((incSock = ufSocket.accept(
            (struct sockaddr*)&incomingAddr, &incomingSize)) == SOCKET_ERROR)
        {
            throw std::runtime_error("Could not accept new connections.");
        }

        ufSocket.send(DATA, DATA_LENGTH, 0);
        ufSocket.recv(buffer, sizeof(buffer), 0);
    }
}
```

Figure 12. Server application via Unified Framework.

Another example in figure 13 demonstrates client code that is using the non-OOP style. In this case, the debugging output to the console is turned on automatically.

In both cases, the location can be set to any value for public communication, except for IP 255.255.255.255 and port 65535, which indicates private communication.

```
#include "uf.h"

#include <iostream>
#include <cstdio>
#include <stdexcept>

char buffer[8192];

int main() {
    socket_t socket = UFNetwork::POSIX::socket(AF_INET, SOCK_STREAM, 0);

    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = 0;
    if (UFNetwork::POSIX::connect(socket,
        (struct sockaddr*)&addr, sizeof(addr)) == SOCKET_ERROR)
    {
        throw std::runtime_error("Could not connect to server.");
    }

    std::cout << "Connected!\n\n";
    UFNetwork::POSIX::send(socket, DATA, DATA_LENGTH, 0);
    UFNetwork::POSIX::recv(socket, buffer, sizeof(buffer), 0);
}
```

Figure 13. Client application via Unified Framework.

Additionally, the program in figure 14 can be used to calculate the private node ID.

```
#include <iostream>
#include "uf.h"

int main() {
    uint8_t* privateNodeId = new uint8_t[UFNetwork::Node::NODE_ID_LENGTH];
    try {
        UFCrypto::calculatePrivateNodeIdFromKeyFiles(privateNodeId);
    } catch (const std::exception& e) {
        std::cout << "Error occurred: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Figure 14. Application example that calculates the private node ID.

To run a Rendezvous node, the following program in figure 15 can be used.

```
#include <iostream>
#include "uf.h"

int main() {
    UFNetwork::Rendezvous node;
    UFUtil::DebugProcessor* logger = new UFUtil::DebugLogger;
    node.setDebugProcessor(logger);

    try {
        node.runServer();
    } catch (const std::exception& e) {
        std::cout << "Error occurred: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Figure 15. Application that hosts a Rendezvous node.

This is essentially the full extent of the API that should be used directly by application developers.

3.5.2 Injection

For the method of injection, the injector program, as well as the shim, need to be built from source code. The Microsoft Detours dependency also needs to be compiled according to the steps given in the files: *detours_build_x86/x64.txt*. After this, the *detours-example.sln* file can be opened to build the projects. The building may fail the first time since there is a need to link the shim to the Unified Framework, so the files *uf.dll* and *uf.lib* need to be placed on the appropriate paths according to the error log. After the second build attempt, the files *inject.exe* and *ufshim.dll* will be produced as a result. Along with the main Framework file *uf.dll* and the applicable encryption keys in the same directory, it is possible to now perform the injection. To do that, the keyword *inject.exe* must be prepended to the desired command line. For example, if one wants to launch a Python script normally, it would be enough to launch the following command: *python script.py*. To pass the script through an injector, one needs to execute this command: *inject.exe python script.py*.

Unfortunately, it is not possible to generate application or secret keys using the injection method, so it would be required to generate them natively beforehand. The same applies to the ability to calculate the private node ID.

4 Evaluation of the Framework

The completed project can now be evaluated using different methods, which include various software engineering principles, use cases and similar projects. Then developments for the future will be outlined.

4.1 Software engineering principles

First, the codebase design can be assessed as to whether it follows the reasonable software engineering principles and patterns, as well as which ones exactly. There are several sets of principles. The ones chosen here for evaluation are: GRASP (General Responsibility Assignment Software Patterns) by Craig Larman [51], Clean Code by Robert C. Martin [52], SOLID by Robert C. Martin and Michael Feathers [53] [54], as well as select ones from GoF (Gang of Four) by Erich Gamma et al [55].

4.1.1 GRASP

Table 2. Evaluation of Unified Framework classes according to GRASP.

UFCrypto	is an Information Expert at: • updating the IV (Initialization Vector) during encryption and decryption • storing the temporary AES key during private communication has High Cohesion because: • it manages only cryptographic operations for the most part (an exception is reading keys from and saving keys to the file system)
UFNetwork	–

UFNetwork.Node	is the Creator of: • an object that is the instance of or derives from UFNetwork.DHT is an Information Expert at: • keeping track of whether the current node has been initialized • knowing what is the current node's role • loading application and secret keys from files on the file system has High Cohesion because: • it is responsible for all operations that initialize a node
UFNetwork.UPnP	is the Creator of: • the UFNetwork object that contains a temporary data buffer is an Information Expert at: • storing the names of available network cards • storing the IP addresses of the NAT device behind network cards • storing the HTTP URL of the UPnP API of NAT devices has High Cohesion because: • it is responsible only for UPnP-related operations

UFNetwork.DHT	is the Creator of: • the UFNetwork object that contains a temporary data buffer is an Information Expert at: • maintaining a list of *bootstrap* DHT nodes • storing the last found node ID, in case one has been found has High Cohesion because: • it is responsible only for storing the state and executing the code for the DHT client exhibits Polymorphism because: • the *mergeNodes(...)* function is available in multiple forms, depending on the collection type
UFNetwork.DHTAnnounce	is the Creator of: - (subclass of UFNetwork.DHT) is an Information Expert at: - (subclass of UFNetwork.DHT) • keeping track of the nodes located within distance of the node ID that should be announced • keeping track of the node ID for a started announcing operation has High Cohesion because: • it is responsible only for storing the state and executing additional announcing operations for the DHT client

UFNetwork.Rendezvous	is the Creator of: • the UFNetwork object that contains a temporary data buffer is an Information Expert at: • keeping track of node IDs as well as locations of servers that performed the *REGISTER* command has High Cohesion because: • it only manages the state and code that is related to a Rendezvous node
UFNetwork.POSIX	is the Creator of: • the UFNetwork object that contains a temporary data buffer for a socket • the UFCrypto object that manages cryptographic information of a socket is an Information Expert at: • interacting with virtual socket handles • keeping track of socket *contexts*, which include various information, such as the real socket handle and objects that contain data buffers and cryptographic information • knowing what is the current *connection behavior* of the virtualized socket has High Cohesion because: • it is designed to only be a container for virtual POSIX-like networking API methods

UFNetwork.POSIX	exhibits Polymorphism because: • it supports both OOP and non-OOP socket creation, which produces the same overall result • different code is being executed depending on the ”connection behavior” currently set
UFUtil.DebugProcessor	is an Information Expert at: • storing the custom minimum severity of debugging logs that should be processed is a Controller for: • accepting debugging events and processing them in some way has High Cohesion because: • it is only related to processing debugging events exhibits Polymorphism because: • it is an abstract class that allows performing one abstract operation in different ways
UFUtil.DebugLogger	is an Information Expert at: - (subclass of UFUtil.DebugProcessor) is a Controller for: • accepting debugging events and outputting them to the console has High Cohesion because: • it is only related to logging debugging events

Additionally, overloaded constructors in various classes and default arguments manifest Polymorphism.

The sequence of object interactions for the following public operations constitutes Low Coupling:

- generation of application and secret keys and saving them to the file system
- calculation of the private node ID based on the *app.key* and *secret.key* files
- all operations that virtualize the POSIX networking API except *accept(...)*, *connect(...)*, *send(...)*, *recv(...)*

A large number of other operations constitute *tight coupling* due to the requirement of interconnecting many components.

The classes UFUtil.HTTPParser and UFUtil.XMLParser (not mentioned in Table 2) exhibit Pure Fabrication or Indirection patterns in relation to the class UFNetwork.UPnP, because the UPnP API needs to support HTTP and XML parsing.

The class UFUtil.Bencode (not mentioned in Table 2) exhibits Pure Fabrication or Indirection patterns in relation to the classes UFNetwork.DHT and UFNetwork.DHTAnnounce, because the Bencode format is used for the Mainline DHT protocol.

The entire codebase is using the Protected Variations pattern due to the requirement of being portable to various operating systems.

4.1.2 Clean Code

In this section, the code is analyzed according to the Clean Code software engineering principles.

Meaningful names:

- ”Use Intention-Revealing Names” – YES/NO, when it comes to class instance variables, the names were indeed chosen such that they reveal intent. However, local variables inside methods might have been named more arbitrarily in some cases.

- ”Avoid Disinformation” – YES/NO, due to the more lax naming of local variables in methods, they might disinform the reader. Also, the *nodeId* class variable in UFNetwork.DHT does not refer to the same kind of ”node ID” as the variable *lastFoundNodeId*. In other cases, I have tried to avoid disinformation in class variables.
- ”Make Meaningful Distinctions” – YES/NO, in some ways, distinctions between names are made. For example, the variables *nonce* and *throwawayNonce* in the UFNetwork.DHT class. In other places, like the *nodes* variable of the same class, the contents of the pair aren't properly labeled because of the implementation of the *std::pair* standard library class. There are, however, ways to make this more distinctive, but this has not been done.
- ”Use Pronounceable Names” – YES, all the names were chosen specifically so that they can be pronounced in the English language.
- ”Use Searchable Names” – YES, the names that can be considered important are well-searchable on the GitLab repository page, for example.
- ”Avoid Encodings” – YES, encodings were not used during the creation of names.
- ”Avoid Mental Mapping” – NO, some mental mappings have been used, for example, in the local variables of the methods of the UFNetwork.DHT class. It is worth refactoring such methods in the future.
- ”Class Names” – YES/NO, for the most part, the class names are nouns that describe a particular module or technology. In one case, the word *Processor* is included, in the class UFUtil.DebugProcessor.
- ”Method Names” – YES/NO, method names start from a verb in most cases. Some methods that start from ”*get*” are not accessors (getters), but more complex functions.
- ”Don't Be Cute” – YES, humorous names are not used within the codebase of the project.
- ”Pick One Word per Concept” – YES, similar verbs for each concept were used in function names where applicable.

- ”Don’t Pun” – YES/NO, as with the previous principle, the verbs were made consistent. However, in some cases, the verb *get* means different things, as already explained. In most cases, they were specifically chosen not to be misleading.
- ”Use Solution Domain Names” – YES, classes and methods are tied to the specific technologies used, as this was the structure chosen for the design. Hence, the names used are for the most part solution-specific as well.
- ”Use Problem Domain Names” – YES/NO, problem domain names are used within the virtual methods of the UFNetwork.POSIX class, but on the other hand, most other names are focused on the specific solution proposed, not the problem.
- ”Add Meaningful Context” – NO, such context was not in mind during development, even though some places would greatly benefit from it. I believe this should be taken into account when refactoring in the future.
- ”Don’t Add Gratuitous Context” – YES, as was described in the comment of the previous principle, context was not in mind, which means that there is no gratuitous context either.

Verdict: The basics have been applied when it comes to properly naming things in the codebase, but more advanced techniques have not been used. There is definitely refactoring potential when it comes to names.

Functions:

- ”Small!” – YES/NO, most functions and methods are not small in size, with the exception of some such as *calculatePrivateNodeIdFromKeyFiles* in UFCrypto, for example.
- ”Do One Thing” – NO, methods are not split in such a way that they are only designed to do one (small) thing. The strategy was different, to offload technical procedures to a few functions which can do it well, but that resulted in long functions.
- ”One Level of Abstraction per Function” – NO, methods using different high-level and low-level procedures are intermixed and not separated. This is partially due to the initial strategy of having a small number of classes per component of the Framework.

- ”[Avoiding] Switch Statements” – NO, switch statements are used in the code, including in the middle of functions. This can probably be minimized with further refactoring.
- ”Use Descriptive Names” – YES, in some classes such as UFCrypto, method names are exclusively descriptive, and in other classes, almost all of them are as well.
- ”Function Arguments” – YES/NO, a lot of methods have 2-3 or more arguments, although some with 0-1 arguments exist too. The solution to this is not immediately obvious, perhaps if other principles are applied, then this one will become trivial to apply as well.
- ”Have No Side Effects” – YES/NO, some methods such as *bootstrapForAnnounce* in UFNetwork return a value, even though it is optional and can be considered as a side effect of the method. A large number of other methods do not have any other effects other than what they were supposed to do.
- ”Command Query Separation” – NO, this principle has not been explicitly used, and some methods such as *generateKeysToDisk* in UFCrypto perform multiple-key generation using boolean flags, which is not the best solution in terms of readability. The method *generateSecretKeys* in the same class checks the arguments for *null* values to choose which keys to generate, which is not the best design.
- ”Prefer Exceptions to Returning Error Codes” – YES/NO, the codebase uses a mix of exceptions and return codes. Perhaps it would be better to only use exceptions.
- ”Don’t Repeat Yourself” – YES/NO, even though duplication was meant to be avoided during the design of the methods, some places such as the UFNetwork.DHT class contain quite a few duplicated code snippets. This is because I have not found a reasonable way to split them. With enough refactoring efforts, it should be possible to get rid of duplication.
- ”Structured Programming” – NO, *goto* statements are used for optimization purposes. Some methods also use multiple *return* statements, which means that these principles of structured programming are not used.
- ”How Do You Write Functions Like This?” – YES/NO, methods were written in a way to first get the code to work properly, but on the other hand, not enough refactoring has been done to split them up into smaller functions.

Verdict: The code is complete enough to the point where the primary structure of methods has been defined, and the base implementation is written, but not much more than that. The existing ”clean” patterns can be used as inspiration for future refactoring.

Comments:

- ”Comments Do Not Make Up for Bad Code” – NO, since comments have been used for certain parts of the code that are complex. A different strategy should probably be used to achieve ”cleaner” code.
- ”Explain Yourself in Code” – NO, as with the previous principle, the code was not necessarily structured in a readable way, but just so that the API makes sense.
- ”Good Comments”
  - ”Legal Comments” – NO, such comments have not been used.
  - ”Informative Comments” – YES/NO, for example, the comment indicating the reasons for the *marker* variable in the *initPrivateContextServer* method of UFCrypto can be considered informative. Otherwise, there are simply not enough comments in the first place to say that this principle has been used extensively.
  - ”Explanation of Intent” – NO, comments explaining intent are not present, but could be added.
  - ”Clarification” – YES/NO, some clarifying comments have been used, for example in the *socketInternal* method of UFNetwork.POSIX, but for the most part there are not enough comments in the first place to say that this principle has been used extensively.
  - ”Warning of Consequences” – NO, such comments are not present in the code.
  - ”TODO Comments” – YES, whenever there are some features that clearly need to be worked on, some TODO comments have been added.
  - ”Amplification” – YES, such comments have been used in the few places that require them, for example in the *acceptTraverseRdvu* and *connectTraverseRdvu* methods of UFNetwork.POSIX.
  - ”Javadocs in Public APIs” – NO, Doxygen comments have not been used anywhere in the codebase.

- ”Bad Comments”
  - ”Mumbling” – NO, such comments have not been used. The existing comments were meant to point at a specific part of the code and be concrete.
  - ”Redundant Comments” – NO, there are no redundant comments, because there are not many comments in the first place.
  - ”Misleading Comments” – NO, the existing comments do match up with the code.
  - ”Mandated Comments” – NO, since the need to use Doxygen comments has not been mandated by anybody.
  - ”Journal Comments” – NO, changelogs are not present in the codebase.
  - ”Noise Comments” – YES/NO, such comments are not used in the code, except perhaps the *generateKeysToDisk* function of the UFCrypto class, where the comments are meant to separate logical pieces of the function.
  - ”Scary Noise” – NO, there are no misleading redundant comments.
  - ”A Comment When You Can Use a Function or a Variable” – NO, such cases do not seem to be present.
  - ”Position Markers” – YES, this style was used in the UFNetwork.POSIX class implementation due to the need to separate OOP and non-OOP code, but perhaps there is a better way.
  - ”Closing Brace Comments” – NO, closing brace comments are not used.
  - ”Attributions and Bylines” – NO, attributions are not used within comments.
  - ”Commented-Out Code” – YES, there is some commented out code because I considered it important.
  - ”HTML Comments” – NO, all the comments are in plain text.
  - ”Nonlocal Information” – NO, all comments concern local code.
  - ”Too Much Information” – NO, all comments are short (maybe too brief).
  - ”Inobvious Connection” – YES, some comments may need more in-depth clarification themselves.
  - ”Function Headers” – NO, these are not used.

    - ”Javadocs in Nonpublic Code” – NO, because Doxygen comments are not used anywhere, let alone in non-public code.

Verdict: There is not a big amount of comments in the Unified Framework’s codebase, which might be both good and bad in the long term. Also, there are no unnecessary comments which would need to be cleaned up.

Formatting:

- ”Vertical Formatting” – NO, some files contain a large number of lines of code, so that every class can represent a separate module of the Framework as closely as possible. Perhaps it is reasonable to split the code definition files (.cpp) even further.
    - ”The Newspaper Metaphor” – YES/NO, some files have been written specifically with this principle in mind. However, in some classes such as UFNetwork.POSIX, a lot of small wrapper functions are present in the bottom, plus other similar cases.
    - ”Vertical Openness Between Concepts” – YES, vertical spacing is used to separate blocks of code for a more cohesive understanding.
    - ”Vertical Density” – YES/NO, related code is grouped, however not always is the density so explicit.
    - ”Vertical Distance”
        - ”Variable Declarations” – YES, variables are declared and defined usually near the places where some code needs to use them.
        - ”Instance variables” – YES, in the header files the instance variables are separated from methods closer to the top, Java-style.
        - ”Dependent Functions” – YES/NO, related functions are usually located close to each other, where the caller is on the bottom and the callee is on top (which is the style of the C and C++ languages), however, this is not always the case.
        - ”Conceptual Affinity” – YES, functions that are similar or closely related are located close to each other in the code.

  - ”Vertical Ordering” – YES/NO, as with ”Dependent Functions”, the ordering is for the most part such that the caller is on the bottom and the callee is on top.

- ”Horizontal Formatting” – YES, most lines are at most 100 characters long. In very few cases, the maximum width gets to 120.
  - ”Horizontal Openness and Density” – YES, this principle has been explicitly used when writing the code.
  - ”Horizontal Alignment” – YES, this principle has been explicitly used when writing the code.
  - ”Indentation” – YES, the code is indented using a specific pattern.
    - ”[Avoiding] Breaking Indentation” – YES/NO, as it is possible that in some small cases the indentation may be inconsistent.
  - ”Dummy Scopes” – NO, because, in my opinion, a loop without curly braces can act as a statement.
- ”Team Rules” – NO, since there were no other developers who would oversee the project.

Verdict: The code formatting can be considered good enough already, with the only problem being long classes and functions.

Objects and Data Structures:

- ”Data Abstraction” – YES/NO, most of the public instance variables are abstracted away, but this is not the case for private classes or variables. However, some public data is accessible directly and not via getters/setters, so refactoring is needed to fix this flaw.
- ”Data/Object Anti-Symmetry” – YES, since not everything is an object, even though the codebase is mainly object-oriented, this principle applies.
- ”The Law of Demeter” – NO, considering that the project uses tight coupling and some methods are static, so other classes can call them.

  - ”[Avoiding] Train Wrecks” – YES, it does not seem like chained calls to methods of different classes grouped using local variables are present.
  - ”[Avoiding] Hybrids” – NO, some structures are hybrid, especially if the class is private. This was done due to ease of use but may need refactoring.
  - ”Hiding Structure” – NO, not enough structure is hidden.
- ”Data Transfer Objects” – NO, the DTO pattern is not used.
  - ”Active Record [as a Data Structure]” – NO, because Data Transfer Objects are not used.

Verdict: The idea was to have a separate class per module. This may be what made this part of the Clean Code principles harder to achieve. It may or may not be worth it to abstract data away in the private sections in a similar way as it is in the public sections.

Error Handling:

- ”Use Exceptions Rather Than Return Codes” – YES/NO, the codebase uses a mix of exceptions and return codes. Perhaps it would be better to only use exceptions.
- ”Write Your Try-Catch-Finally Statement First” – YES/NO, this principle is applied in some places, but not in all.
- ”Use Unchecked Exceptions” – YES, because C++ does not have checked exceptions anyway.
- ”Provide Context with Exceptions” – YES, all exceptions that are meant to be passed to the target application are meant to be informative.
- ”Define Exception Classes in Terms of a Caller’s Needs” – NO, because the exception class used is only ”*std::runtime_error*”. Custom exception classes are not used.
- ”Define the Normal Flow” – NO, a lot of exceptions in the project require *catch*ing.
- ”Don’t Return Null” – NO, in some places *null*s are being returned.
- ”Don’t Pass Null” – NO, *null*s are extensively used to pass into functions.

Verdict: The currently available error handling structures can indicate whether a problem has occurred and even show where, but there is not really a system designed to process errors in an advanced way.

Boundaries:

- ”Using Third-Party Code” – YES/NO, some standard library code is wrapped around in custom interfaces, but not all, so there may be default exceptions from standard library code.
- ”Exploring and Learning Boundaries” – NO, learning tests are not used, because the libraries imported into the project have a fairly straightforward API.
- ”Using Code That Does Not Yet Exist” – YES/NO, even though the project used a bottom-up approach to development, this pattern was still used in some cases, for example, the UFNetwork.DHT class.
- ”Clean Boundaries” – YES/NO, imported code is wrapped around with separate classes, for example, UFUtil.XMLParser. However, UFCrypto is quite dependent on OpenSSL.

Verdict: Many components of the Unified Framework have been written from scratch and there is not a lot of third-party code. It is not clear if one needs to apply these principles if the eventual goal is to have a completely self-sustained codebase.

Classes:

- ”Class Organization” – YES/NO, classes are organized such that constants come first, then instance variables, then methods. This is especially the case for *private* blocks.
  - ”Encapsulation” – YES, *private* and *public* sections in header files are split according to the encapsulation principle. In case it is needed, *protected* sections are used as well.
- ”Classes Should Be Small!” – YES/NO, most classes are designed to only represent one module which usually has a set of responsibilities only for its own uses. An

exception to this might be the usage of UFNetwork.Node variables in UFCrypto and possibly some other places.

  - ”The Single Responsibility Principle” – NO, classes are for the most part designed to have high cohesion, but the classes are quite large and have many responsibilities (but still within the realm of the module).
  - ”Cohesion” – YES/NO, the methods of classes are usually designed to operate on the instance variables only of that class, however, some exceptions apply.
  - ”Maintaining Cohesion Results in Many Small Classes” – NO, the classes are quite large and functions can be long and have many local variables, which means this principle does not apply.

- ”Organizing for Change” – YES/NO, some classes such as UFNetwork.DHT are complete enough that they do not require changes, so the subclass UFNetwork.DHTAnnounce uses inheritance instead. However, the project classes were considered to be predetermined, so the codebase might not be completely organized for changes.
  - ”Isolating from Change” – NO, the codebase was written with specific technologies in mind, so classes are not isolated from change.

Verdict: The class structure is not well-adapted for big changes. It is also not clear if it is worth applying the principles above, at least before applying the ones that are more straightforward for refactoring first.

Systems:

- ”Separate Constructing a System from Using It” – NO, different modules are constructed or initialized on-demand, so this principle does not apply. On the other hand, this may reduce the number of initialized components on startup. But during runtime, this might lead to slowdowns, so there is work to be done in relation to this principle.
  - ”Separation of Main” – NO, this principle has not been applied, but it is worth exploring in the future.

  - ”Factories” – NO, factories are not used in the project, but it does seem like a good idea to use them.
  - ”Dependency Injection” – NO, dependency injection is also not used. The number of modules or components in the Unified Framework is quite fixed and tied to its core goals, however, some components are still optional, for example, UFNetwork.UPnP. It might be worth it to explore Dependency Injection.

- ”Scaling Up” – YES/NO, it depends on whether the concerns of the system are separated properly. From what it seems now, the different outer classes like UFNetwork, UFCrypto and UFUtil can be considered as blocks that separate concerns, while the inner classes have the actual implementation. On the other hand, the Framework has *tight coupling*, so it may depend on the direction of ”scaling up” that determines whether this principle has been applied.
  - ”Cross-Cutting Concerns” – NO, for example, the UFNetwork.DebugProcessor abstract class is intertwined with the code.

- ”Java Proxies” – YES/NO, the *bridge library* or *shim* might be considered as a certain proxy, but otherwise, this pattern is not used in the main code.

- ”Test Drive the System Architecture” – NO, the code cannot be ”test-driven” by itself efficiently.

- ”Optimize Decision Making” – YES/NO, it might be possible that there is room for changes when different decisions need to be made about the Unified Framework in the future, but on the other hand, the architecture is ”set in stone” and not meant to be changed heavily.

- ”Use Standards Wisely, When They Add Demonstrable Value” – YES/NO, ”industry” standards have not been heavily used in this project, as it was meant to be a proof-of-concept of something potentially innovative.

- ”Systems Need Domain-Specific Languages” – NO, domain-specific languages are not used.

Verdict: The Unified Framework itself can indeed be described as a quite complex system, so it seems like applying these principles would greatly benefit it. For example, using Factories and initializing all the necessary components on launch.

Emergence:

- ”Simple Design Rule 1: Runs All the Tests” – YES/NO, the project uses *tight coupling*, so it is hard to test it by unit. Integration testing in the form of demo applications is available, but unfortunately, while those test most of the code, they do not test all of the edge cases.
- ”Simple Design Rules 2–4: Refactoring” – YES/NO, the code has been refactored several times to make its structure easier to understand, however, it is not refactored to perfection. There is room for improvement and further refactoring.
- ”No Duplication” – YES/NO, there was an attempt to avoid duplication, however, in some places such as the UFNetwork.DHT and UFNetwork.DHTAnnounce classes there is quite a bit of duplication.
- ”Expressive” – NO, the code has not been refactored further after the final proof of concept has been completed, because the more important point was to confirm that the design of the API is appropriate, not so much that the implementation is perfect.
- ”Minimal Classes and Methods” – YES/NO, the number of classes is indeed low, and this is by design. However, it is not simple to avoid having many methods due to the complexity of the system.

Verdict: The system is indeed using some principles of Emergent/Simple Design, but not to the fullest extent. It should be easier to continue refactoring to achieve all of them, even though the project is considered to be finished now. Nonetheless, by this point, it is possible to confirm that the class structure and implementation of the Framework are at least somewhat appropriate because the project can be tested and verified that it works.

Concurrency is only used in one place within the codebase in the UFNetwork.Rendezvous class. It contains a *mutex*, that represents a synchronized section for closing and reopening a socket. This implementation can be improved in such a way, that synchronization is not required in the first place. The UFNetwork.DHT class and its subclasses might require additional improvements in the form of concurrency or threading to speed up the Framework's initialization process.

To conclude, there is potential to refactor the codebase of the Unified Framework much more and it can be one of the future developments. It would probably not be possible to easily apply all of the principles, because of the established structure of the codebase, but it is should be possible to apply a large part of them.

4.1.3 SOLID

The evaluation of each of the SOLID principles is given as follows:

Single Responsibility Principle – this principle has not been fulfilled, because the classes are divided per technology or module used, not per responsibility.

Open-Closed Principle – some classes within the Unified Framework are easily inherited from and their functionality extended, but some are not, for example, UFNetwork.Rendezvous or UFNetwork.POSIX. The open-closed principle is only partially fulfilled.

Liskov Substitution Principle – the instances where the class UFNetwork.DHT is used can be replaced with the UFNetwork.DHTAnnounce class, as well as the instances of UFUtil.DebugProcessor can be replaced with UFUtil.DebugLogger. These are the only two instances currently where subclasses are used, thus the Liskov Substitution Principle has been fulfilled.

Interface Segregation Principle – public interfaces of the classes are designed to be cohesive with what they are supposed to do, so the Interface Segregation Principle is being applied.

Dependency Inversion Principle – the project has been developed with the help of the *bottom-up approach*, so a lot of technologies were already in mind before the codebase design phase. What follows from this is that the Dependency Inversion Principle has not been fulfilled.

4.1.4 GoF

The following GoF patterns are present in the main codebase of the Unified Framework:

Creational Patterns:

- *Factory Method*, the *initializeNode()* method in the UFNetwork.Node class can be considered as a Factory Method because it sets the *dhtCache* variable to different instances of either the UFNetwork.DHT class or its descendant. It is being delegated the creation of a new object with the needed configuration.
- *Singleton*, this pattern is used in the *initChildContext(...)* method of the UFNetwork.POSIX class. In case the *contexts* list does not exist, it is created. All subsequent calls to this method reuse the already existing list.

Structural Patterns:

- *Adapter*, this pattern is used for example by the UFUtil.XMLParser class (not mentioned in Table 2). The class *wraps* around a third-party library *pugixml* to provide its own interface, effectively acting as an Adapter to provide the same functionality in a different way.
- *Facade*, this pattern is used everywhere throughout the codebase. The class UFNetwork.Rendezvous uses it with the method *runServer()*, the class UFNetwork.DHT with the *bootstrap()* method, etc. The goal was to simplify the API for the target developer of an application.

Behavioral Patterns:

- *Observer*, this pattern is partially used for setting debugging event processors inside multiple classes. The *setDebugProcessor(...)* method, as well as some constructors, allow one to set this to an instance of any descendant of UFUtil.DebugProcessor. If this has not been done, then event processing does not take place. The reason for the partial and not full use of this pattern is that only one DebugProcessor can be set per class. The design could surely be improved by allowing classes to process events using multiple DebugProcessors.
- *Strategy*, it is possible to identify the different ”connection behaviors” inside the UFNetwork.POSIX class as different Strategies, however, these are not separate

classes but just a flag inside the class. Therefore, this pattern has only partially been used.

4.2 Use cases of the Framework

As mentioned previously, the Unified Framework should have a variety of different use cases. Several target groups of developers who the Unified Framework would be suitable for have been identified.

The first group is users who may want to set up various services on the Internet right from their homes. Then, various home servers and network projects can be remotely accessed from any arbitrary remote device, as long as they are within the same application context. This allows regular users to effectively contribute computation resources to the Internet.

The second group are developers who deal with the field of real-time communication between users, for example, to exchange text messages or launch video conferences, and want the users' privacy to be preserved.

The third group is IoT developers who would rather not set up Cloud-based infrastructure to manage customer devices since the user's home network can take on the burden of hosting instead.

Another group is people who would like to adapt existing client-server source code or applications for P2P use cases. This should be trivial with the bridge library that the Unified Framework provides.

The final group is developers who want to provide services based on a proprietary code-base to transform it into a public networked "black box".

Other target groups may exist. Using the Unified Framework for P2P applications can cut development time and the need to learn or integrate various P2P technologies since the Framework is only one single comprehensive implementation. Also, due to it using *native code*, performance can be increased compared to non-native solutions. With enough refactoring, it should be possible to make the Unified Framework lightweight enough to run successfully on very low-powered embedded devices.

Being able to establish a direct connection between arbitrary devices can be considered essential to moving the Internet in the direction of further decentralization. If such a scenario suddenly became commonplace, it would lead to transformational changes of the Internet.

Unfortunately, the Unified Framework will not be suitable for those who want to implement P2P clients inside web browsers, as the support for that has not been added yet.

4.3 Overview of similar frameworks

Similar projects that work on similar layers already exist, so this section lists other related libraries and compares them to the Unified Framework.

4.3.1 JXTA

The first related project is JXTA, which started development by Sun Microsystems in 2001. It also works on top of the Transport layer like the Unified Framework and provides some similar features, such as Peer Discovery, data encryption, but also grouping of peers, relaying, file sharing, authentication, authorization, etc. Metadata is transferred between peers (nodes) in the form of XML messages [56].

JXTA is designed to be language and platform-independent, as is the Unified Framework, and is divided into six subprotocols: Peer Discovery Protocol (PDP), Peer Information Protocol (PIP), Peer Resolver Protocol (PRP), Pipe Binding Protocol (PBP), Endpoint Routing Protocol (ERP) and Rendezvous Protocol (RVP). The choice of protocols is up to the developer, and some can be optional [57].

Two main interfaces, JxtaSocket and JxtaBiDiPipe are available through which communication between the nodes can take place [57], where the former is a subclass of the *java.net.Socket* class, at least on its native language Java, and the latter is a more high-level interface [58].

Like Unified Framework node IDs, JXTA has the concept of ”peer ID”, which seems to work in a similar way. C and C++ language compatibility is also included.

Node discovery is achieved in JXTA using the Peer Discovery Protocol (PDP), which should work on LAN by default, but to discover peers on the global Internet, the usage of additional ”rendezvous peers” is needed (not to be confused with the Rendezvous nodes of the Unified Framework, the terminology varies). There is no DHT-like directory in JXTA, different nodes just relay information to each other [59]. Some hardcoded ”rendezvous peers” were provided, but the servers are currently inaccessible [60].

JXSE, the original Java implementation of JXTA [61], did not support NAT traversal as of 2010, and instead relayed information through other peers, similar to TURN [62] [63].

Also, unlike the Unified Framework, this library was not designed for compatibility with POSIX sockets in mind, as it includes many more public APIs [64].

Unfortunately, the development was reportedly stopped in 2010 [65] but has continued in some capacity in 2013 [66].

4.3.2 freedom.js

The second project described is freedom.js. Unlike the Unified Framework, it is a web-based framework that is meant to be used inside browsers. It completely redefines the style of how applications should be made with its modular API design [67].

For the transport protocol, it uses WebRTC, which is embedded into many common web browsers [68]. This technology runs on top of UDP and provides NAT traversal capabilities by default via STUN, as well as relaying via TURN [69].

The primary design choice of freedom.js is modularity. Rather than selecting a few essential technologies that should be used for all applications, the authors made it possible to interchange modules used for transport, storage and identification. This means that technically even the WebRTC protocol can be replaced by a developer with another, or an existing implementation improved, and this would still be possible via the native freedom.js API. These modules are meant to be reused across applications, similar to how a package registry like npm[1] provides third-party ready-to-use libraries for developers.

DHT is used in this framework not for node discovery, but for data storage purposes. To identify other users, it seems that XMPP or OpenID are used, and it is possible to add more sources using different modules. Data encryption features are provided through WebRTC, which is mandatory with that protocol.

[1] https://www.npmjs.com/

The architecture is different from the one in the Unified Framework, which determines a preset amount of technologies to be used as a base, and all others are meant to run on top of those without the intervention of the Framework. This is in contrast to freedom.js, which allows one to swap out even the essential parts with other ones.

The authors have also introduced a browser extension named ”freedom.js Manager”, which is designed to bypass restrictions imposed by the web browser’s security model, such as storage space or computation time. However, it does not seem like this extension is available in the Google Chrome Web Store[1], the Firefox Add-ons page[2] or the project’s GitHub repository[3].

Nevertheless, it may be beneficial to use this project for the development of brand new browser-based P2P applications, but it is unsuitable for standalone applications which can run directly on a computer or mobile device.

4.3.3 libp2p

The final related solution is named libp2p, which is described by the authors as a ”modular peer-to-peer networking stack” [70]. Just like the previous project, this one uses a system of different components. Here they are divided into groups: Transports, Multiplexers, Secure Channels, Peer Discovery, Peer Routing, Content Routing, NAT Traversal and Pubsub. Libp2p is located above the Transport layer of communications, which is the same placement as in JXTA and the Unified Framework. Peers (nodes) are likewise marked with an identifier, which is derived from the public key. The public key itself can use a variety of different asymmetric encryption algorithms [70].

Node discovery is provided either using mDNS for local networks (LAN) or Kademlia DHT (the same kind that Mainline DHT is based on). NAT Traversal and relaying are performed via custom protocols that are similar to STUN and TURN respectively.

Libp2p is designed to run both inside standalone applications, as well as in a web browser. Currently, it supports the following programming languages: JavaScript, Go and Rust, where the Go implementation is the most complete one [71]. Not all modules have been

[1] https://chrome.google.com/webstore/search/freedom.js
[2] https://addons.mozilla.org/en-US/firefox/search/?q=freedom.js
[3] https://github.com/freedomjs/freedom

developed in all languages, so the project can be considered unfinished[1].

Unlike the Unified Framework, libp2p requires manual instantiation and usage of the components required [72].

4.3.4 Summary

To summarize, even though the Unified Framework has been developed independently, it inadvertently overlaps with other P2P frameworks using many of the same technologies. This is probably either because the low-level architecture of P2P applications is pretty straightforward or this kind of design makes the most sense. The Unified Framework does however offer some benefits that others don't, namely, the compatibility with the POSIX API and the heavy use of the *Facade* software engineering pattern to abstract away most underlying details for the application developer, sacrificing modularity and extensibility that may ultimately not be needed.

4.4 Future developments

Throughout the analysis, multiple points concerning the challenges of the current version of the Unified Framework have been identified. To address them, the following improvements can be made in the future:

- Right now the Framework only supports the TCP Transport layer protocol, but the UDP protocol can be added in relatively easily. Using UDP can be more efficient compared to TCP in some applications, and methods to traverse the NAT with UDP already exist. One of them is STUN (Session Traversal Utilities for NAT) [73].
- An issue might be related to the usage of Symmetric NATs and VPN services, where the NAT traversal protocol of the Framework can fail [74]. In this case, it is worth adding additional traversal algorithms, as well as designing an extra system for relaying data efficiently similar to TURN (Traversal Using Relays around NAT) [75], where no other option is available.
- To increase stability, it is possible to add self-correcting, resilient and robust protocols to recover from non-fatal errors. This includes the creation of a priority system for ”connection behaviors” in the UFNetwork.POSIX class.

[1] As of May 2022.

– Since Certificate Authorities (CAs) are not used within the Framework architecture, it might be worth implementing a ”web of trust” [76] instead, which should strengthen security.

– Adding support for IPv6 can make the Unified Framework more future-proof.

– To avoid congestion of Rendezvous nodes, the DAT system from a reviewed research paper [39] can be applied on top of the existing DHT without the use of ”gateways” to eliminate unnecessary *bottlenecks*. Also to avoid loss of service due to the inability of finding a node identifier, a background thread should periodically *reannounce* nodes to the DHT.

– Adding distributed storage support can increase the number of use cases for the Framework and fit in well within the current network architecture. Another node type can be added, whose responsibility would be to store chunks of data from users for certain periods of time.

– WebRTC and WebSockets support can be added for web browsers to be able to connect to servers hosted via the Unified Framework as clients.

– The project should be ported to Unix-based systems including mobile devices to increase relevance and usability.

– To make sure that the Unified Framework can be used in ”air-gapped” intranets, another node role ”DHT server” should be added. It would provide services in place of the public one from BitTorrent.

– Decoupling the Unified Framework from third-party libraries such as OpenSSL and pugixml can make the project's code fully independent.

– Refactoring the code further while taking more software engineering principles and patterns into account can make the code ”cleaner”, more understandable to new developers.

– The support for an additional automatic *port forwarding* protocol can be added alongside UPnP. NAT-PMP [77] is another one that should not take long to implement.

– An alternative for node discovery on local networks (using mDNS or UDP Multicast) can reduce confusion and provide better options for debugging.

– Node initialization can be made to perform as soon as the application is launched to also reduce confusion.

– Node configuration (for example, cached DHT bootstrap nodes or the private node ID to connect to) can be saved into local files on the file system. This way the GUI window when communicating privately can be avoided and the number of single points of failure reduced.

– Debugging flexibility can be increased by allowing multiple DebugProcessors per class and by defining a structure of event constants.

– Some methods in the UFNetwork.DHT and UFNetwork.DHTAnnounce classes can be made asynchronous to increase the application initialization speed.

5 Conclusion

Before the result as part of this thesis has been achieved, developers of P2P applications had experienced various problems. For example, they were not able to achieve direct connections between nodes on the Internet in a compatible way. The Unified Framework has been proposed as a solution that abstracts away all the technologies that are necessary for direct connections to get successfully established. To begin the work on the Framework, first, various background concepts have been explained. Additionally, the relevant literature has been reviewed. Then, the design of the codebase has been established. Afterwards, the network architecture has been put in place. It was then possible to test the Framework whether it works correctly. After the testing has been done, the Unified Framework has been evaluated according to various software engineering principles and patterns. Also, it has been explained which target groups can benefit the most from the Framework. The Framework has then been compared with other similar projects that already exist. Finally, some suggestions for the future of the Framework in terms of development have been proposed.

References

[1] A. Oram, "Peer-to-peer: Harnessing the power of disruptive technologies," in, 1st ed. O'Reilly, 2001, ch. A Network of Peers: Peer-to-Peer Models Through the History of the Internet, pp. 3–14, ISBN: 978-0-596-00110-0.

[2] K. Aberer and M. Hauswirth, "An Overview on Peer-to-Peer Information Systems," p. 14, 2002.

[3] J. L. Eppinger, "TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem," p. 8, 2005.

[4] P. H. Nguyen, "A Literature Review about Peer to Peer Protocol," p. 3, 2020.

[5] G. Maier, F. Schneider, and A. Feldmann, "NAT Usage in Residential Broadband Networks," in *Passive and Active Measurement*, N. Spring and G. F. Riley, Eds., vol. 6579, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 32–41, ISBN: 978-3-642-19259-3 978-3-642-19260-9. DOI: 10.1007/978-3-642-19260-9_4.

[6] A. Moallem, "Home Networking: Smart but Complicated," in *Human-Computer Interaction. Applications and Services*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, and M. Kurosu, Eds., vol. 8512, Cham: Springer International Publishing, 2014, pp. 731–741, ISBN: 978-3-319-07226-5 978-3-319-07227-2. DOI: 10.1007/978-3-319-07227-2_70.

[7] *What Is My IP Address - See Your Public Address - IPv4 & IPv6*, https://whatismyipaddress.com/, Accessed: 2022-05-08.

[8] S. S. H. Hajjaj and K. S. M. Sahari, "Establishing remote networks for ROS applications via Port Forwarding: A detailed tutorial," *International Journal of Advanced Robotic Systems*, vol. 14, no. 3, May 2017, ISSN: 1729-8814, 1729-8814. DOI: 10.1177/1729881417703355.

[9] ICANN, *Beginner's Guide to Domain Names*, Brochure, 2010. [Online]. Available: https://www.icann.org/en/system/files/files/domain-names-beginners-guide-06dec10-en.pdf.

[10] J. Schlamp, J. Gustafsson, M. Wählisch, T. C. Schmidt, and G. Carle, “The Abandoned Side of the Internet: Hijacking Internet Resources When Domain Names Expire,” in *Traffic Monitoring and Analysis*, M. Steiner, P. Barlet-Ros, and O. Bonaventure, Eds., vol. 9053, Cham: Springer International Publishing, 2015, pp. 188–201, ISBN: 978-3-319-17171-5 978-3-319-17172-2. DOI: 10.1007/978-3-319-17172-2_13.

[11] M. R. Parwez, M. Akbar, S. Haider, and M. S. Javaid, “DNS propagation delay: An effective and robust solution using authoritative response from non-authoritative server,” in *2010 2nd IEEE International Conference on Information Management and Engineering*, Chengdu, China: IEEE, 2010, pp. 150–153, ISBN: 978-1-4244-5263-7. DOI: 10.1109/ICIME.2010.5477485.

[12] T. Hancock, “Programming languages and ”lock-in”,” *Free Software Magazine*, Apr. 18, 2008, Accessed: 2022-05-08. [Online]. Available: http://freesoftwaremagazine.com/articles/programming_language_lock_in/.

[13] *Operating System Market Share Worldwide | Statcounter Global Stats*, https://gs.statcounter.com/os-market-share, Accessed: 2022-05-08.

[14] A. E. Kwame, E. M. Martey, and A. G. Chris, “Qualitative assessment of compiled, interpreted and hybrid programming languages,” *Communications on Applied Electronics*, vol. 7, no. 7, pp. 8–13, 2017, ISSN: 2394-4714.

[15] T. Kawato, M. Higashino, K. Takahashi, and T. Kawamura, “Proposal of e-Learning System integrated P2P Model with Client-Server Model,” in *2019 International Conference on Electronics, Information, and Communication (ICEIC)*, Auckland, New Zealand: IEEE, Jan. 2019, pp. 1–6. DOI: 10.23919/ELINFOCOM.2019.8706472.

[16] H. S. Oluwatosin, “Client-server model,” *IOSR Journal of Computer Engineering*, vol. 16, no. 1, pp. 67–71, Feb. 2014, ISSN: 2278-8727, 2278-0661. DOI: 10.9790/0661-16195771.

[17] H. Zimmermann, “OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection,” *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 425–432, Apr. 1980, ISSN: 0096-2244. DOI: 10.1109/TCOM.1980.1094702.

[18] M. Jahajee, A. Katlana, N. Khare, and P. Diwakar, "OSI Model," *International Journal of Engineering Sciences & Management Research*, Oct. 2015, ISSN: 2349-6193.

[19] A. L. Russell, "OSI: The Internet That Wasn't," *IEEE Spectrum*, Jul. 29, 2013, Accessed: 2022-05-08. [Online]. Available: `https://spectrum.ieee.org/osi-the-internet-that-wasnt`.

[20] A. N. A. Ali, "Comparison study between IPV4 & IPV6," *International Journal of Computer Science Issues*, vol. 9, no. 3, May 2012, ISSN: 1694-0814.

[21] S. Hogg, "You Thought There Was No NAT for IPv6, But NAT Still Exists," *Infoblox Blogs*, Dec. 28, 2021, Accessed: 2022-05-08. [Online]. Available: `https://blogs.infoblox.com/ipv6-coe/you-thought-there-was-no-nat-for-ipv6-but-nat-still-exists/`.

[22] Y. Zhangyi, M. Yan, F. Baker, H. Xiaohong, and Z. Xiaodong, "The implementation of NAT66 and the solutions of multi-homing in NAT66 environment," in *2009 2nd IEEE International Conference on Broadband Network & Multimedia Technology*, Beijing, China: IEEE, Oct. 2009, pp. 608–613, ISBN: 978-1-4244-4590-5. DOI: `10.1109/ICBNMT.2009.5347847`.

[23] *IANA IPv4 Special-Purpose Address Registry*, `https://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml`, Accessed: 2022-05-08.

[24] *Service Name and Transport Protocol Port Number Registry*, `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`, Accessed: 2022-05-08.

[25] J. Lv, W. Tang, and H. Zhang, "A TCP-based Asymmetric NAT Traversal Model for P2P Applications," in *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, Harbin City, Heilongjiang, China: IEEE, Dec. 2012, pp. 883–886, ISBN: 978-1-4673-5034-1 978-0-7695-4935-4. DOI: `10.1109/IMCCC.2012.212`.

[26] *Understanding Different NAT Types and Hole-Punching*, `https://dh2i.com/kbs/kbs-2961448-understanding-different-nat-types-and-hole-punching/`, Accessed: 2022-05-08.

[27] *Private Address Ranges - IBM Documentation*, `https://www.ibm.com/docs/en/networkmanager/4.2.0?topic=translation-private-address-ranges`, Accessed: 2022-05-13.

[28] K. Egevang and P. Francis, "The IP Network Address Translator (NAT)," RFC Editor, Tech. Rep. RFC1631, May 1994. DOI: `10.17487/rfc1631`.

[29] B. Cui, Q. Zhang, X. Zhang, and T. Guo, "Research on UPnP Protocol Security of Gateway Device," in *Advances on Broad-Band Wireless Computing, Communication and Applications*, L. Barolli, F. Xhafa, and J. Conesa, Eds., vol. 12, Cham: Springer International Publishing, 2018, pp. 450–458, ISBN: 978-3-319-69810-6 978-3-319-69811-3. DOI: `10.1007/978-3-319-69811-3_41`.

[30] *WANIPConnection:2 Service*, UPnP Forum, p. 49, Sep. 2010.

[31] Z. Hu, "NAT traversal techniques and peer-to-peer applications," in *HUT T-110.551 Seminar on Internetworking*, Citeseer, 2005, pp. 04–26.

[32] *Windows Sockets 2*, `https://docs.microsoft.com/en-us/windows/win32/winsock/windows-sockets-start-page-2`, Accessed: 2022-05-08.

[33] *Using Sockets and Socket Streams*, `https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/NetworkingTopics/Articles/UsingSocketsandSocketStreams.html`, Accessed: 2022-05-08.

[34] *Socket(7) Linux User's Manual*, Mar. 2021.

[35] L. Besaw, *Berkeley UNIX† System Calls and Interprocess Communication*, Jan. 1987.

[36] *Machine Code*, `https://icarus.cs.weber.edu/~dab/cs1410/textbook/1.Basics/machine.html`, Accessed: 2022-05-08.

[37] A. Turcotte, E. Arteca, and G. Richards, "Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language," p. 32, 2019. DOI: `10.4230/LIPICS.ECOOP.2019.16`.

[38] J. Berdajs and Z. Bosnić, "Extending applications using an advanced approach to DLL injection and API hooking," *Software: Practice and Experience*, pp. 567–584, 2010, ISSN: 00380644, 1097024X. DOI: `10.1002/spe.973`.

[39] M. B. M. Kamel, P. Ligeti, A. Nagy, and C. Reich, “Distributed Address Table (DAT): A Decentralized Model for End-to-End Communication in IoT,” *Peer-to-Peer Networking and Applications*, vol. 15, no. 1, pp. 178–193, Jan. 2022, ISSN: 1936-6442, 1936-6450. DOI: 10.1007/s12083-021-01221-3.

[40] K. Matsuoka and T. Suzuki, “Blockchain and DHT Based Lookup System Aiming for Alternative DNS,” in *2020 2nd International Conference on Computer Communication and the Internet (ICCCI)*, Nagoya, Japan: IEEE, Jun. 2020, pp. 98–105, ISBN: 978-1-72815-800-6. DOI: 10.1109/ICCCI49374.2020.9145989.

[41] O. Ohwo, O. Awodele, and O. Yewande, “An Understanding and Perspectives of End-To-End Encryption,” vol. 08, no. 4, pp. 1086–1094, Apr. 2021, ISSN: 2395-0056, 2395-0072.

[42] K. Lin and Z. Jiang, “Using a Dynamic Domain Name System (DDNS) Technology to Remotely Control a Building Appliances Network,” in *International MultiConference of Engineers and Computer Scientists*, vol. 1, 2017, ISBN: 978-988-14047-3-2.

[43] *DHT Protocol*, http://bittorrent.org/beps/bep_0005.html, Accessed: 2022-05-08.

[44] K. L. Busbee and D. Braunschweig, “Programming Fundamentals – A Modular Structured Approach, 2nd Edition,” in. Dec. 15, 2018, ch. Modular Programming, p. 133.

[45] R. Hartson and P. Pyla, “Bottom-Up Versus Top-Down Design,” in *The UX Book*, Elsevier, 2019, pp. 279–291, ISBN: 978-0-12-805342-3. DOI: 10.1016/B978-0-12-805342-3.00013-8.

[46] I. Grigorik, “High Performance Browser Networking: What every web developer should know about networking and web performance,” in. O’Reilly, Oct. 29, 2013, ch. Transport Layer Security (TLS), pp. 47–52, ISBN: 978-1-4493-4476-4.

[47] National Institute of Standards and Technology, “The Keyed-Hash Message Authentication Code (HMAC),” National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST FIPS 198-1, Jul. 2008, NIST FIPS 198–1. DOI: 10.6028/NIST.FIPS.198-1.

[48] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC Editor, Tech. Rep. RFC2104, Feb. 1997. DOI: 10.17487/rfc2104.

[49] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings. Conference on Software Maintenance 1990*, San Diego, CA, USA: IEEE Comput. Soc. Press, 1990, pp. 290–301, ISBN: 978-0-8186-2091-1. DOI: 10.1109/ICSM.1990.131377.

[50] T. Ziadé, "Python microservices development: Build, test, deploy, and scale microservices in Python," in. Packt Publishing, Jul. 25, 2017, ch. Discovering Flask, pp. 33–64, ISBN: 978-1-78588-111-4.

[51] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Pearson, Oct. 20, 2004, p. 736, ISBN: 978-0-13-148906-6.

[52] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Pearson, Aug. 1, 2008, ISBN: 978-0-13-235088-4.

[53] R. C. Martin, "Design principles and design patterns," *Object Mentor*, p. 34, 2000.

[54] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st ed. Pearson, Sep. 10, 2017, pp. 57–59, ISBN: 978-0-13-449416-6.

[55] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Eds., *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Addison-Wesley, Nov. 10, 1994, ISBN: 978-0-201-63361-0.

[56] N. Maibaum and T. Mundt, "JXTA: A technology facilitating mobile peer-to-peer networks," in *International Mobility and Wireless Access Workshop*, IEEE, 2002, ISBN: 978-0-7695-1843-5. DOI: 10.1109/MOBWAC.2002.1166946.

[57] R. Mekki and R. Fezza, "A Sample Chat Application Based on JXTA," *Journal of Applied Sciences*, vol. 9, no. 21, pp. 3912–3916, Oct. 2009, ISSN: 1812-5654. DOI: 10.3923/jas.2009.3912.3916.

[58] M. Abdelaziz, "Demystifying Pipes, JxtaSockets, JxtaMulticastSocket, and JxtaBiDiPipes," *java.net Weblogs*, Aug. 23, 2005, Accessed: 2022-05-13. [Online]. Available: https://web.archive.org/web/20091106092646/http://weblogs.java.net/blog/2005/08/23/demystifying-pipes-jxtasockets-jxtamulticastsocket-and-jxtabidipipes.

[59] J. D. Gradecki, “Mastering JXTA building Java peer-to-peer applications,” in. Wiley Pub., 2002, pp. 31–37, ISBN: 978-0-471-42936-4.

[60] S. Oaks, B. Traversat, and L. Gong, “JXTA in a nutshell,” in. O’Reilly, 2002, pp. 20–24, ISBN: 978-0-596-00236-7.

[61] *The JXTA JXSE Open Source Project on Open Hub*, `https://www.openhub.net/p/jxta-jxse`, Accessed: 2022-05-13.

[62] I. Neto and F. Reverbel, “Using JXTA for Firewall Traversal in Distributed CORBA Applications,” in *6th SBC Workshop on Free Software*, vol. 5, 2005.

[63] J. Verstrynge, “Practical JXTA II: Cracking the P2P puzzle,” in. DawningStreams, 2010, p. 110, ISBN: 978-1-4461-3956-1.

[64] *JXTA Java™ Standard Edition v2.5: Programmers Guide*, Sun Microsystems, Inc., p. 57, Sep. 10, 2007.

[65] J. Verstrynge, “Latest News,” *JXSE Wiki Home Page*, Dec. 31, 2010, Accessed: 2022-05-13. [Online]. Available: `https://web.archive.org/web/20120820145752/http://kenai.com/projects/jxse/pages/LatestNews`.

[66] *P2p - Why has JXTA been abandoned? Any alternatives out there? - Stack Overflow*, `https://stackoverflow.com/a/10342234`, Accessed: 2022-05-13.

[67] W. Scott, R. Cheng, A. Krishnamurthy, and T. Anderson, “Freedom.js: An Architecture for Serverless Web Applications,” University of Washington Computer Science and Engineering, Seattle, Washington, Tech. Rep. UW-CSE-13-05-03, 2013.

[68] N. M. Al-Fannah, “One leak will sink a ship: WebRTC IP address leaks,” in *2017 International Carnahan Conference on Security Technology (ICCST)*, Madrid: IEEE, Oct. 2017, pp. 1–5, ISBN: 978-1-5386-1585-0. DOI: `10.1109/CCST.2017.8167801`.

[69] B. Garcia, F. Gortazar, L. Lopez-Fernandez, M. Gallego, and M. Paris, “WebRTC Testing: Challenges and Practical Solutions,” *IEEE Communications Standards Magazine*, vol. 1, no. 2, pp. 36–42, 2017, ISSN: 2471-2825, 2471-2833. DOI: `10.1109/MCOMSTD.2017.1700005`.

[70] *Introduction to libp2p*, `https://max-inden.de/static/introduction-to-libp2p.pdf`, Accessed: 2022-05-13.

[71] *Implementations - libp2p*, `https://libp2p.io/implementations/`, Accessed: 2022-05-13.

[72] *Go-libp2p/chat.go at master · libp2p/go-libp2p · GitHub*, `https://github.com/libp2p/go-libp2p/blob/master/examples/chat-with-rendezvous/chat.go`, Accessed: 2022-05-13.

[73] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, D. Wing, R. Mahy, and P. Matthews, "Session Traversal Utilities for NAT (STUN)," RFC Editor, Tech. Rep. RFC8489, Feb. 2020. DOI: `10.17487/RFC8489`.

[74] Y. Wang, Z. Lu, and J. Gu, "Research on Symmetric NAT Traversal in P2P applications," in *2006 International Multi-Conference on Computing in the Global Information Technology - (ICCGI'06)*, Bucharest: IEEE, Aug. 2006. DOI: `10.1109/ICCGI.2006.60`.

[75] T. Reddy, A. Johnston, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," RFC Editor, Tech. Rep. RFC8656, Feb. 2020. DOI: `10.17487/RFC8656`.

[76] A. Mathew, "Can Security be Decentralised?: The Case of the PGP Web of Trust," in *Socio-Technical Aspects in Security and Trust: Proceedings of 11th International Workshop, STAST 2021*, ser. Lecture Notes in Computer Science, 2021.

[77] S. Cheshire and M. Krochmal, "NAT Port Mapping Protocol (NAT-PMP)," RFC Editor, Tech. Rep. RFC6886, Apr. 2013. DOI: `10.17487/rfc6886`.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Igor Podgainõi

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "A Unified Framework for Peer-to-Peer Applications", supervised by Toomas Klementi and Gunnar Piho
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

13.05.2022

[1] The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.